Interdisciplinary Variability Modeling and Performance Analysis for Long-Living Automation Systems

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von

Matthias Kowal

geboren am 02.11.1987

in Soest

Eingereicht am: 29.09.2017

Disputation am: 01.12.2017

1. Referentin: Prof. Dr. Ina Schaefer

2. Referent: Prof. Dr. Matthias Tichy

2017

Abstract

In this day and age, automation systems have to deal with differing customer needs, environmental requirements and multiple application contexts. Automation systems have to be variable enough to satisfy all of these demands. The development and maintenance of such highly-customizable systems is a challenging task and becomes increasingly more difficult considering multiple involved engineering disciplines and long lifetimes, which is characteristic for industrial systems of the automation domain. Software product line engineering provides developers with fundamental concepts to manage the variability of such systems. However, these concepts are not established in the domain of automation systems. In addition, the involvement of multiple engineering disciplines poses a threat to existing SPL techniques. This thesis contributes novel approaches to improve the development and maintenance of software-intensive automation product lines. In total, three major contributions are made, spanning across the complete design phase of an automation system. (1) The feature modeling process is improved by detecting hidden dependencies between interrelated feature models from separate engineering disciplines. Furthermore, hidden dependencies and occurring defects in the feature models are explained in a user-friendly manner. (2) A model-driven development approach is introduced consisting of UML models, which are extended with delta modeling to manage variability in the automation product line. The models encompass information that is needed to automatically derive and analyze a performance model. (3) Subsequently, an efficient family-product-based performance analysis is proposed for the previously derived UML models that is vastly superior compared to common product-based approaches. All of these techniques have been evaluated using multiple case studies, with one being a real-world automation system.

Zusammenfassung

In der heutigen Zeit sehen sich Automatisierungssysteme mit einer steigenden Komplexität konfrontiert. Einzelne Kunden haben unterschiedliche Ansprüche an das System und ebenso müssen Umweltbedingungen der verschiedenen Betriebsumgebungen sowie abweichende Anwendungsgebiete bei der Entwicklung eines Automatisierungssystems berücksichtigt werden. Diese Komplexitätsaspekte werden unter dem Stichwort Variabilität zusammengefasst. Ein Automatisierungssystem muss in der Lage sein, sämtliche Anforderungen zu erfüllen. Die Entwicklung und Wartung dieser Systeme wird jedoch durch die stetig wachsende Variabilität und eine potentiell lange Lebensdauer immer schwieriger. Zusätzlich sind an dem Entwicklungsprozess eines Automatisierungssystems mehrere Ingenieursdisziplinen beteiligt. Die Techniken aus dem Bereich der Software-Produktlinienentwicklung bilden Lösungen, um die Variabilität beherrschbar zu machen. In der Automatisierungstechnik sind diese Techniken weitgehend unbekannt und durch den interdisziplinären Charakter oft nicht ausreichend. Daher werden in dieser Dissertation neue Ansätze entwickelt und vorgestellt, die auf die Domäne der Automatisierungstechnik zugeschnitten sind. Insgesamt leistet diese Dissertation folgende drei wissenschaftlichen Beiträge: (1) Die Entwicklung von Feature-Modellen wird durch die Detektion von verborgenen Abhängigkeiten, die zwischen Feature-Modellen der unterschiedlichen Ingenieursdisziplinen existieren, verbessert. Gleichzeitig liefert der vorgestellte Algorithmus die Erklärung für die Existenz dieser Abhängigkeiten. Dieses Konzept wird auf weitere Defekte in Feature-Modellen ausgeweitet. (2) Einen modell-basierten Ansatz zur Entwicklung eines Automatisierungssystems. Der Ansatz basiert auf Modellen aus der UML, die mit Hilfe der Delta Modellierung Variabilität abbilden können. Zusätzlich sind die Modelle mit Informationen über Performance Eigenschaften angereichert und erlauben die automatische Ableitung eines Performance-Modells. (3) Eine effiziente Performance Analyse von allen Varianten des Automatisierungssystems, die auf den zuvor abgeleiteten Performance-Modellen basiert. Alle Beiträge wurden mit Fallstudien evaluiert. Eine Fallstudie repräsentiert ein reales Automatisierungssystem.

Acknowledgements

Many people supported me in different ways across the years and I would like to express my gratitude to the most influential persons within the following lines.

First and foremost, I wish to thank my supervisor Ina Schaefer for giving me the opportunity to do my Ph.D. at her institute. She encouraged me to take this step and her continuous guidance is the reason why I was able to finish this work. Whether I was stuck with my research or just needed some advice, she always took the time to lend me an ear and discuss possible solutions. I thank my reviewer Matthias Tichy for fruitful discussions about new research ideas and especially taking the effort to review this thesis with all thereby connected duties.

Besides Ina and Matthias, I wish to sincerely thank all other professors and researchers that shared their experience and guided me during the last years. In particular, I would like to thank Thomas Thüm for his help to identify and refine promising ideas in the topic of interdisciplinary variability modeling. Our discussions and your critical paper reviews helped me a lot in the difficult process of successfully writing papers. Mirco Tribastone and Max Tschaikowski introduced me to the field of software performance engineering. They were never weary to explain new concepts in a comprehensible way. I could not have wished for better partners in our shared projects. A special thanks goes to Joachim Axmann for his continuous support across many years of my Ph.D. and even beyond my time at the university.

I thank my colleagues at the TU Braunschweig for one of the best working atmospheres imaginable. I had so much fun with you guys attending conferences, discussing work-related topics or just enjoying the lunch break together.

The evaluation of my contributions required a considerable amount of engineering effort. I am grateful to all students, who chose to conduct their own theses under my supervision. The majority of the tool support has been implemented by Sofia Ananieva, Timo Günther and David Lorefice. Many results would have been impossible without their dedication.

I also wish to thank my family for their never-ending support. My mother and father always believed in my abilities and stood behind my decisions. They took so many problems off my mind and enabled me to follow this path of education to the end. I thank my brother for lighting my interest in the domain of computer science.

Many thanks go to my friends from Braunschweig, Wolfenbüttel and Herzfeld. Finally, I am grateful to my partner Vanessa, who has accompanied me through all ups and downs during the complete duration of my thesis. She provided me with the strength and motivation to accomplish this achievement.

Contents

1 Introduction

Current market dynamics force companies to develop countless individualized products in order to meet all the demands of their customers [BBO+12, PBvdL05]. The automation domain is not immune to this request for mass customization of mass-produced products [VHFF+15]. Thus, modern automation systems simultaneously exist in many variants to comply with the differing customer requirements and application contexts. Additionally, they must operate over several decades and adjust to changing environmental conditions, such as functional, technical or legal requirements during their lifetime [BBO+12]. As a result, companies face the challenge to deal with this increasing variability, while keeping the costs low for themselves and the quality high for their customers.

A key success factor in the development of such a variability-intensive automation system is reuse. Engineers should reuse already developed system artifacts whenever it is possible. In the literature, reusability is often achieved by applying the concept of product line development [KA11, PBvdL05, CE05, CN02, KCH+90]. Its advantages are well recognized in combination with software systems such as the Linux kernel [TLSSP11], however they are not yet state-of-the-art for the development of automation systems [BBO+12]. KePlast and KeMotion are two examples for which product line techniques have successfully been applied to the automation domain [LEGP15]. KePlast is a platform for the automation of injection molding machines and KeMotion is a control system for robotics.[1] A direct application of software product line techniques is not sufficient for automation systems, since their development often involves multiple stakeholders from various disciplines such as mechanical, electrical and software engineering. Thus, the development must consider not only dependencies within each discipline, but among the disciplines as well [BBO+12]. Additionally, this interdisciplinary aspect has to be addressed on two levels as software product line development is typically divided into problem and solution space [PBvdL05, CE05, KA11]. The problem space is referred to as the set of all valid system specifications and consists of domain-specific concepts and features. A concrete specification requires the selection of features that the desired system should have. The problem space also contains information about invalid feature combinations and default dependencies [CE05]. The solution space on the contrary is referred to as the set of concrete systems and consists of implementation artifacts. Developers can compose these artifacts to create a system implementation based on

[1] `http://www.keba.com/de/home`

the concrete specification. Naturally, there must be a mapping between both spaces available in order to enable this process. The mapping and all defined dependencies in the problem space are also referred to as configuration knowledge [CE05]. This thesis directly contributes to the research focus of product line development in the automation domain across both spaces.

1.1 Research Question

The idea of this thesis is to provide a holistic and efficient modeling approach covering variability and performance aspects across several engineering disciplines during the design time of an automation system. While there have been considerable advances in modeling variability in software product lines [CE05, KA11, KCH+90, BSRC10], existing approaches lack the functionality to deal with the interdisciplinary nature of automation systems [BBO+12]. Furthermore, the analysis of non-functional performance properties such as throughput, utilization and average response time is a crucial task during the development of an automation system, e.g., to identify optimization potential in its layout. Existing work focuses on expressive modeling and analysis of performance in systems with few or just one variant [BDIS04, BKR09, CM02, BM05]. Approaches that can efficiently execute such quality analyses are already scarce in the general area of software product lines [GS11, TAK+14] and thus also not available for automation systems. Hence, there is a strong need to develop an interdisciplinary variability modeling concept and combine it with an efficient performance analysis covering the complete design time of a product line in the automation domain. This ultimately leads to the central research question of this thesis:

How can we efficiently develop variant-rich systems in the automation domain and analyze their performance properties during design time?

In order to provide an adequate answer, it is paramount to solve three problems that are reflected within the following research questions. Finding an answer to each question provides a stepping stone towards an overall improved development process of an automation product line.

- **RQ1:** How can we reuse and adapt existing variability modeling approaches to deal with the interdisciplinary nature of automation systems in the problem space?
- **RQ2:** How can we capture the complete variability of a real-world automation system during its design time in a uniform way considering the solution space?
- **RQ3:** How can we efficiently analyze product lines in the automation domain regarding performance metrics?

Variability modeling is a key aspect and part of two research questions. While **RQ1** focuses on variability in the problem space that captures domain knowledge in an abstract way, **RQ2** deals with variability in the solution space and concrete development artifacts. A combination of both questions provides us with the holistic variability modeling approach for automation systems in problem and solution space. Finally, developing an efficient performance analysis in **RQ3** enables us to evaluate all variants of the automation system with regard to performance metrics such as throughput or utilization. Hence, we can bring both worlds, variability modeling and performance analysis, together and close the research gap specified in the central research question.

1.2 Approach

This work aims at improving model-based automation product line engineering in both problem and solution space. Fig. 1.1 depicts the overall approach developed in this thesis with respect to the defined **RQs** and individual tasks **T1-7** that must be completed to find an answer to each **RQ**. A description of the figure is part of the following paragraphs in which we take a closer look at each of the three contributions.

Variability in the Problem Space (RQ1). Most approaches for variability modeling, in general, and feature modeling [KCH+90] in particular, consider one large model that specifies all features and their valid combinations [HTH13, BSRC10]. Feature models are the de-facto standard for variability modeling in the problem space [PBvdL05, CE05]. However, experience with applying feature modeling to real-world product lines shows that modeling all features within one model does not scale well in terms of visualization, maintainability, and automated analysis, especially with regard to multiple involved disciplines [LEGP15, LFVH13]. Each change in a feature model may lead to a defect or may arbitrarily influence any part of the feature model [BSRC10, TBK09]. Several approaches try to mitigate these problems such as feature model composition, feature model views and feature modeling for multi software product lines. Feature model composition is the process of combining two or more feature models into a larger feature model [RSTS11, HTH13]. The idea is that we can develop feature models of manageable size and combine them on demand. Typically, there are constraints that link features of one feature model with those of other feature models. An alternative approach is that of feature model views [HTH13]. Instead of composing feature models from smaller models, we can define views on top of an existing, possibly larger feature model. Each view is typically specific to a certain group of stakeholders. Both feature model composition and feature model views contribute to what is known as multi software product

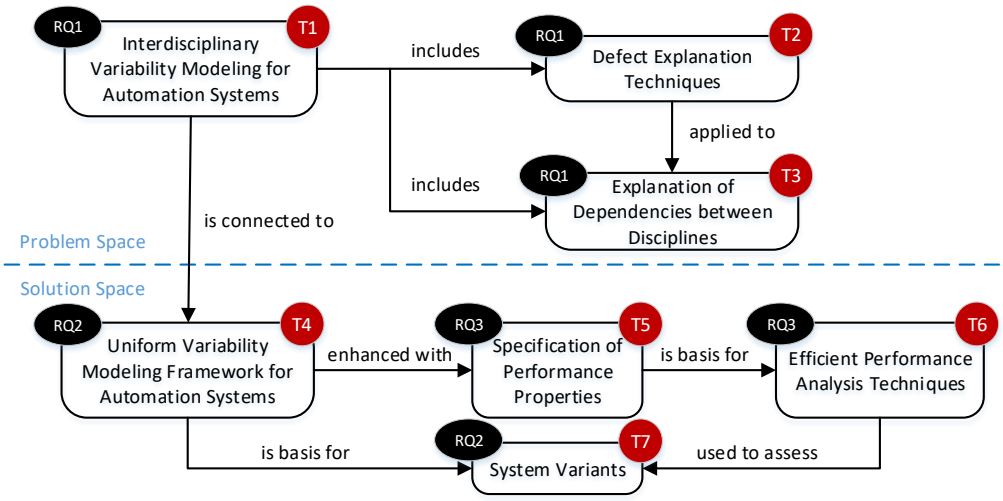


Figure 1.1: Overview of the proposed approach.

lines [HGR12, BRN+13]. That is, we can reuse existing product lines within another product line.

A feature modeling approach that encompasses appropriate measures to deal with the interdisciplinary aspect and supports engineers in the comprehension of defects and dependencies to ensure sound feature models is missing at the moment. Existing approaches either support only the interdisciplinary aspect or provide means to explain defects in one large feature model. The explanation of dependencies between different disciplines is not considered at all in literature.

1. Contribution: We propose improvements to the development and maintenance of feature models tackling these missing aspects. Our approach encompasses support for multiple disciplines by using feature model decomposition **(T1)** and presenting the reasons for possible dependencies to other domains **(T3)**. Consequently, we assist developers in the debugging process of defects in a feature model raising its overall quality **(T2)**. We can apply similar concepts developed in **(T2)** to also fulfill **(T3)** (cf. Fig. 1.1). The improvements are integrated in an existing feature modeling framework and their impact on response times is not perceptible by users. In particular, we operate on propositional formulas that allow efficient computations for feature models and process the obtained results to provide developers with visual feedback.

Variability in the Solution Space (RQ2). Variability is studied extensively in the context of software product lines [PBvdL05]. Existing approaches can be classified in three main directions: annotative (or negative, [VG07]), compositional (or positive, [KAK08]) and transformational [Sch10, HMPO+08]. Annotative approaches consider one model representing all products of the product line. Variant annota-

tions, e.g., using stereotypes in UML models [Gom04] or presence conditions [CA05], define which parts of the model have to be removed to derive a concrete product model. Annotative variability models tend to quickly become very difficult to manage for large software product lines with many variants. Compositional approaches associate model fragments with features that are related to a specific configuration. Throughout the years several composition methods have been proposed such as aspect-oriented or feature-oriented software development [NK08, ABKS13]. However, they can only add functionality to an existing product, and the impact of a feature is limited by the used composition technique. Transformational approaches as the common variability language [HMPO$^+$08] or delta modeling [Sch10] are the most flexible ones, since they allow us to freely add, remove and modify model fragments.

A foundational goal of this thesis is to express the variability and all other relevant information of an automation system across the complete design time. The design time is often comprised of multiple development steps in which each step provides a refinement of system development artifacts [PHAB12].

2. Contribution: Following the well-known separation of concerns principle, we propose a multi-perspective modeling approach with three levels of abstraction. Each modeling perspective represents one development step and is specified by models of the Unified Modeling Language [MG15]. Variability is managed using the delta modeling approach. In combination with a mapping between the individual perspectives, we are able to fully model any automation system and all of its variants **(T4)**. Modeling errors are mitigated with the help of a consistency checking concept. The approach encompasses an automatic generation of system variants, which is necessary to actually control a real-world automation system **(T7)**. The mapping between problem and solution space is realized with an application condition restricting the composition of implementation artifacts [PHAB12]. Hence, we have a holistic modeling approach for automation product lines spanning across both spaces.

Performance Analysis in the Solution Space (RQ3). Variant-rich automation systems are not only complex with respect to the analysis of the variability defined in the problem space (e.g., as in a feature model and defect explanation [BSRC10]). Variability in the solution space poses difficulties for the application of existing analysis techniques, such as type checking, static program analysis, model checking and deductive verification [TAK$^+$14]. A naive solution for this problem involves the analysis of each variant in isolation and is called a product-based analysis. This is not scalable for systems with high variability. Already the generation of all possible variants in a product line is infeasible due to an exponential growth in the worst case with the number of features. Even if the generation of all products is

possible, separate analyses of variants perform numerous redundant computations due to similarities in the individual variants. Several other analysis strategies for product lines have already been identified, i.e. feature- or family-based, which are vastly more efficient compared to the naive approach [TAK+14]. A feature-based strategy analyzes all artifacts connected to a specific feature, while ignoring other features or the dependencies stored in the feature model. Thus, the complexity is reduced to a linear number of analysis tasks, i.e., for every feature. A family-based strategy merges all artifacts in one artificial family model, which also incorporates the knowledge about dependencies in the feature model. Analyzing this artificial family model is often more expensive compared to analyzing an individual variant. However, this analysis is only executed once [TAK+14].

We are confronted with similar problems in order to achieve an efficient combination of our modeling and performance approaches. Most of the existing approaches focus on feature models which are insufficient for a performance analysis in automation systems, since the system behavior is not taken into account [GS11]. Although analyzing the source code avoids this problem, we aim at providing performance estimations during early stages of the development, since we solely focus on the design time. An efficient performance analysis is not available in the literature for this purpose.

3. Contribution: As an initial step, we must provide the necessary performance specifications within our multi-perspective modeling approach **(T5)**. Afterwards, a product-based solution is straightforward possible given the system models. However, we devise an efficient family-product-based performance analysis for automation product lines **(T6)**. It is based on behavioral models of our previous contribution. The models are interpreted as queueing networks underlying continuous-time Markov chains. The construction and analysis of an artificial family model forms the core of this contribution. We can use this concept to acquire the steady-state and to calculate several performance properties such as throughput, utilization, (average) queue length and (average) response time for all variants. The general process is applied to different classes of continuous phase-time distributions comprised of exponential- and Coxian-distributed service times **(T6)**. A comparison to a product-based approach shows the supremacy of our analysis type.

1.3 Structure of this Thesis

An introduction to product lines and our running example in Chapter 2 marks the beginning of this thesis. The focus lies on the development process of a software product line and variability modeling techniques that are used later on in the contribution parts. The running example is a real-world automation system, which we use to validate the applicability of all contributions. Due to the nature of our proposed

approach, we dedicate a whole chapter to each contribution. Chapter 3 describes an algorithm, which is capable of explaining dependencies between multiple feature models as well as typical feature model defects in a similar manner. We continue with the introduction of a modeling framework capturing all relevant information of an automation system during design time in Chapter 4. The framework consists of models from the Unified Modeling Language and is extended with delta modeling as variability management concept. In Chapter 5, we propose the efficient family-product-based performance analysis with the artificial family model. We instantiate the idea two times for queueing networks of different complexity showing its applicability. In addition, each of the three contribution chapters incorporates necessary foundations, an evaluation part and related work relevant for this specific topic. We conclude the thesis and discuss future work in Chapter 6. Additional material within the appendix as well as a list of abbreviations can be found at the end of this thesis.

2 Background

The goal of this chapter is to set the general context for the thesis and to provide an introduction into the main topics common for all later contributions. We describe the basic idea of product lines and especially product lines in the software domain. Then, we explain the concrete engineering process of software product lines in more detail as well as two variability modeling techniques and different analysis strategies. The chapter is completed by a description of a software-intensive automation product line serving as running example throughout the thesis.

2.1 Product Lines

During the last century customer demands have shifted back forth between individualized and standardized products. In the beginning, each product was handcrafted for a specific customer such as a car. However, an increasing population size and its rising wealth demanded a significantly improved production process, since crafting products by hand was no longer able to cope with the ever increasing demand. Probably the most infamous solution in history was the introduction of the assembly line by Henry Ford in his factories. As a result, he was able to produce his vehicles significantly cheaper compared to the handcrafted process enabling a production for the mass market. Hence, *mass production* was born [PBvdL05, CE05].

The main drawback was the lack in diversification and during the decades customers developed a demand for more individualized products tailored to their specific needs. Considering the automotive domain cars for different purposes were introduced such as family, sport or all-terrain vehicles. This trend further continued until today and car manufacturers state that almost no vehicle leaving a factory is identical to another one. In addition to *mass production*, manufacturers now have to deal with *mass customization* [PBvdL05, CE05].

Ultimately a product line is defined in the literature as follows:

Definition 2.1: Product Line

"A product line is a set of systems scoped to satisfy a given market." [CE05]

An application of this definition to the previously used automotive domain results in the historical Ford Model T being a product line or nowadays, e.g., the VW Golf 7. Of course, it is not limited to the automotive industry. Product lines can be

found in many parts of our every day life such as the food and computer industries, e.g., customizing sandwiches or configuring a new computer. The software domain has similar needs in terms of high diversity and a cost-efficient development process leading us to the concept of software product lines.

2.1.1 Software Product Lines

Modern software systems get increasingly more complex due to a rising functionality as well as the immense number of possible end devices in order to satisfy the differing customer requirements. Contrary to this complexity, many devices only have limited resources available making it impossible to deploy the complete software such as in case of embedded systems. Hence, a software systems must be specifically tailored to fulfill all customer wishes while meeting all environment requirements [BCK12, PBvdL05]. Again, we can identify aspects of *mass production* and *mass customization* leading to an analogous definition as in Def. 2.1:

Definition 2.2: Software Product Line

"A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN02]

A *feature* is a characteristic that is visible to the end-user and supports the communication of commonalities and differences between individual products of the SPL [ABKS13]. Consider the modern human machine interfaces in cars as a market segment. An SPL for that domain could have features such as speech recognition, navigation software or wireless connections for the mobile phone. SPLs comprise not only pure software products, e.g. Microsoft Office, but many software-intensive embedded systems as well, e.g., control units in the car or a television.

By combining individual features together, an SPL gives birth to a large number of possible variants of the system that have to be managed. This large variant space poses a threat for the development, maintenance and testing process of an SPL. Traditional engineering approaches are not able to deal with the size and complexity introduced by this variability calling for a specific software product line engineering approach [PBvdL05].

2.1.2 Software Product Line Engineering

The problem of *mass production* is easy to solve in software systems, since software can be copied very quickly. However, the greatest threat lies in the *mass customization* and therefore the variability. Managing this variability is a key aspect of the

SPL engineering to ensure an efficient development process. The SPL engineering is defined as follows:

Definition 2.3: Software Product Line Engineering

"SPL engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization." [PBvdL05]

A *platform* is described as a common structure to enable the creation of complete variants of the SPL. They foster reuse potential while minimizing redundancy in the development. Platforms are not limited to source code, but include requirements and all kinds of models as well [PBvdL05, CN02, CE05].

The SPL engineering is divided into two phases, namely, the *domain engineering* and the *application engineering*. The two phases including their sub-steps are described in the next paragraphs and also depicted in Fig. 2.1.

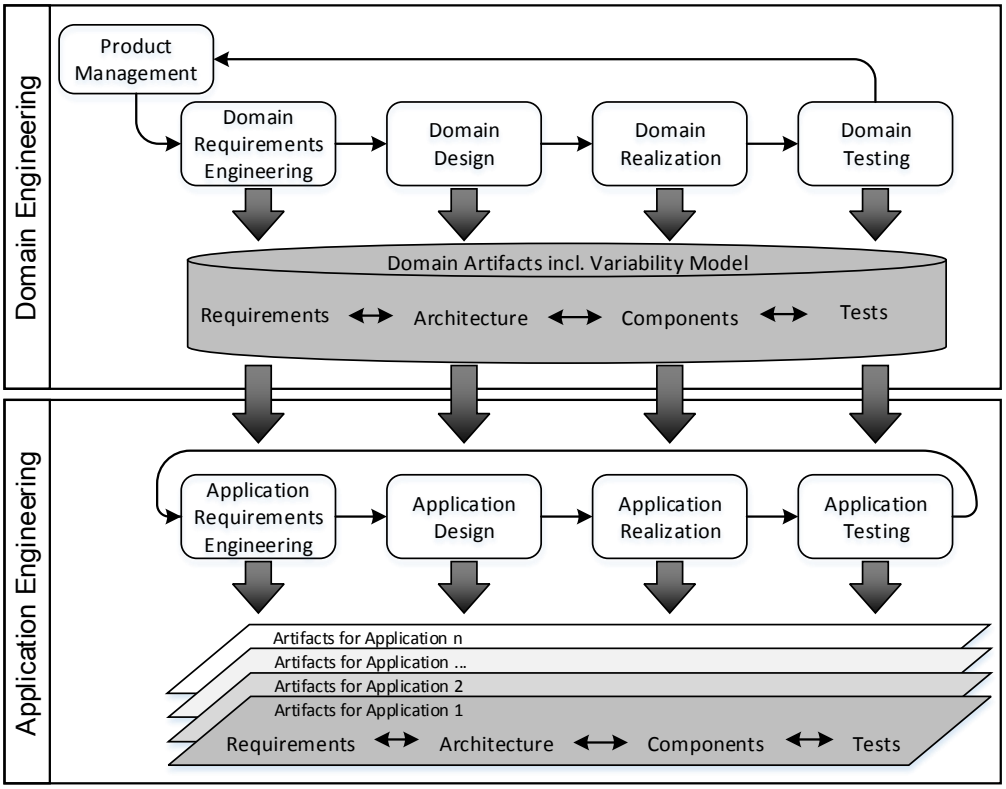


Figure 2.1: The classic SPL engineering process [PBvdL05].

Domain Engineering. Domain engineering establishes the reusable *domain artifacts* spanning from requirements to source code, hence the platforms. The definition is as follows:

Definition 2.4: Domain Engineering

"Domain Engineering is the process of SPL engineering in which the commonality and the variability of the SPL are defined and realized." [PBvdL05]

The process is composed of five sub-phases as depicted in the upper part of Fig. 2.1. First, the **product management** focuses on economical aspects of the SPL and the desired market segment. The scope of the SPL is defined in alignment with the company goals and market strategies. As a result, a *product road map* is created describing the major common and variable features, a release schedule for product variants and if possible a list of existing products and artifacts usable to develop the common platforms for the SPL in the future [PBvdL05].

Second, common and variable requirements of the SPL are identified in the **domain requirements engineering** based on the *product road map*. In contrast to single software systems, these requirements describe the full SPL and therefore all foreseeable variants at that point in time. However, as Fig. 2.1 already depicts the domain engineering is a continuous development process and may have an arbitrary number of iterations. As a result, the derived requirements can be extended or altered in future cycles. A *variability model* is created containing the information of the requirements as well as their commonalities and differences. This model can be on a textual or model-based level.

The third sub-process is called *domain design* and takes the previously derived *variability model* to develop a *reference architecture* by mapping requirements to technical solutions. This architecture serves as a common, top-level structure for all SPL variants enabling the aspect of *mass customization*. During this process the *variability model* is refined to also express internal variability mandatory for the technical solution.

Fourth, the **domain realization** produces *reusable software components* given the defined *reference architecture*. Again, reusability is the key aspect for this process. The designed and implemented software assets should be loosely coupled and provide an adequate interface to maximize the reuse potential. The result of this phase is not a complete product variant of the SPL, but individual components that can be combined to create the desired product. Hence, it also ensures respective configuration mechanisms to select this product given the available components.

The fifth and last sup-process in the domain engineering is **domain testing**. Especially validation and verification of the *reusable software components* is the goal at this point. Thus, the realization is tested against the specification, i.e. requirements

and architecture, and **domain testing** should also detect faults in the individual components. In addition, the developed test artifacts should also be reusable for future iterations. The existing variability model is used to derive potential test artifacts. Contrary to single-system engineering, we have no executable application under test, since this is part of the testing process in the **application engineering**. Only single components are tested at this point.

The complete **domain engineering** process can and often must be repeated to deal with changing requirements, environments or general evolution of the SPL.

Application Engineering. Application engineering is responsible for deriving concrete product variants based on the artifacts created during the domain engineering. It ensures sound binding of the variability to the specific demand of an application. A common definition is as follows:

Definition 2.5: Application Engineering

"Application engineering is the process of SPL engineering in which applications of the product line are built by reusing domain artifacts and exploiting the variability." [PBvdL05]

Except from the product management sub-process in the domain engineering, all described sub-processes have a direct counterpart in the application engineering. The individual domain artifacts serve as respective inputs for these application sub-processes. The lower half of Fig. 2.1 shows their correlation and similar to the previous paragraph, we briefly describe the individual sub-processes.

During the **application requirements engineering** it is mandatory to involve the customer as one of the essential stakeholders, since only the customer knows the desired requirements for the specific application. Ideally, the domain artifacts already capture all customer demands. However, in some cases the defined requirements during the domain engineering are not sufficient for the customer. Hence, the requirements as well as the variability model must be extended to reflect them. Ultimately, we have a complete requirement specification for the particular application.

In the next step, the **application design**, an instance of the reference architecture is invoked to form the concrete application architecture. Necessary application-specific changes are combined with the reference architecture saving time compared to single-system development, since the complete reference architecture or certain parts can be reused multiple times for any number of applications. The output is an architecture fulfilling all specified requirements.

Finally, the **application realization** produces a fully-fledged and executable application based on the architecture and software artifacts. Most crucial part is

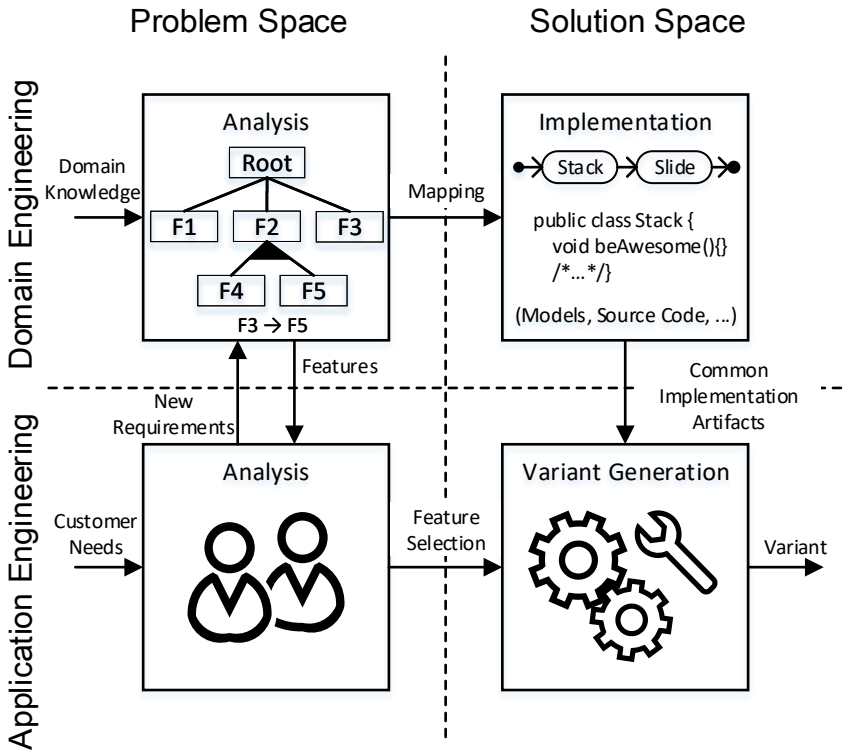


Figure 2.2: Problem and Solution Space in the context of SPL engineering [KA11, ABKS13, CE05].

the selection and configuration of the software components.

The **application testing** focuses to improve the quality of the created application and verifies as well as validates it against the specification. Again, the test artifacts can be derived from the domain testing process enabling a high reuse potential. Ideally, we have a *traceability* mechanism to map requirements to software components and test cases making this sub-phase a lot easier. However, in SPL engineering it is often not possible to test all variants and combinations that can be produced in the application engineering due to limited resources and the exponentially growing variant space [KSS13, LPKS14, ER11].

2.1.3 Problem and Solution Space

SPL engineering is typically divided into the *problem space* and the *solution space*, which is also depicted in Fig. 2.2 [CE05]. The problem space consists of domain-specific abstractions describing the requirements of a software system as well as its intended behavior. The top-left corner in Fig. 2.2 basically is a summarized version of several domain engineering sub-phases previously shown in Fig. 2.1. Again, we emphasize that the results are documented in terms of features that play an important role in the first contribution of this thesis. The solution space solely contains

implementation-oriented abstractions, such as models or source code artifacts (cf. top-right corner in Fig. 2.2). Features in the problem space are mapped to artifacts in the solution space. Hence, a selection of features ultimately results in the selection of the connected implementation components with which the executable application is created (cf. bottom-half of Fig. 2.2). The mapping can have different shapes, which depend on the implementation method and the degree of automation. A straightforward solution with low complexity is a name-based approach due to naming conventions. More sophisticated solutions use constraints that are processable by machines, such as preprocessors [CE05].

In this thesis, we focus on the artifacts created during the domain engineering spanning across the problem and the solution space, hence the complete upper-half of Fig. 2.2. First, we deal with different aspects of a feature model representing the derived variability model (Chapter 3). Second, we provide variability management techniques to model (Chapter 4) and analyze (Chapter 5) implementation-oriented artifacts. The overall goal is to minimize the effort necessary for the development and maintenance of an SPL.

Feature Modeling. In an SPL not all combinations of features are useful or even physically possible (cf. Section 2.2). Feature models are one way to express the intended variability. After their introduction by Kang et al. in 1990, they ascended to be the default modeling concept for SPLs in the problem space [KCH+90, BSRC10, CE05, PBvdL05]. A feature model consists of a hierarchically arranged set of features and has typically a tree-like graphical representation, which is called feature diagram, such as depicted in Fig. 2.3 for a simple computer mouse SPL. Relationships between parent and child features are expressed using the following notations and their semantics (see legend in Fig. 2.3 for graphical notation) [KCH+90, CE05]:

- *Mandatory* – feature must be selected, if the parent is,
- *Optional* – feature is optional,
- *Or* – one or more subfeatures can be selected,
- *Alternative* – only one subfeature can be selected.

Example 2.1: Computer Mouse

For instance, all variants of the computer mouse must have the mandatory features *Left Button*, *Right Button*, *Sensor* and *Connection*. However, we can only choose one sensor type from *Ball* or *Laser*. The connection to a computer is established with an *USB* cable or a wireless *Bluetooth* interface. Some mouse variants also support both connection types. A *Scroll Wheel* is an optional component.

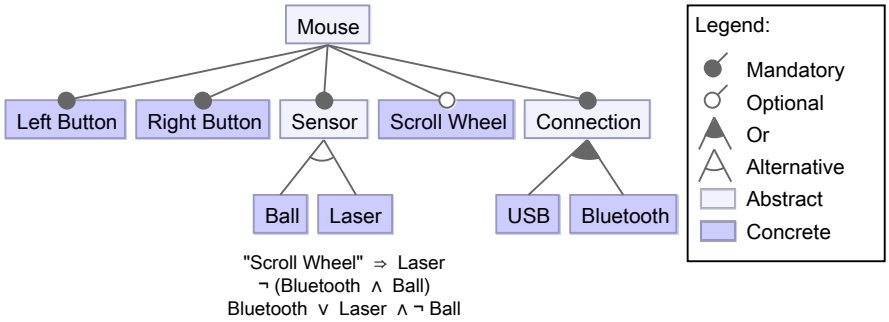


Figure 2.3: Feature diagram for a computer mouse SPL.

Abstract features such as *Mouse*, *Sensor* and *Connection* do not contain realization artifacts and are only used for structural purposes or represent place holders for features planned in the future [TLSSP11]. A *core* feature is present in all variants of the SPL, e.g., *Left Button*. Relationships between features that are not related by a parent-child relationship can be expressed using cross-tree constraints (CTC) written in propositional logic. Fig. 2.3 presents three CTCs below the feature tree. The formulas are not limited to simple require (first CTC) and exclude (second CTC) relationships, but can be arbitrarily complex representing any combination of features, e.g., as depicted in the third CTC [CE05].

At this point, we only introduced the visualization of a feature model that supports developers to build and maintain an SPL. While this representation is easy to understand for humans, propositional formulas are a more efficient representation for analysis purposes [BSRC10].

Propositional Formulas in Feature Models. A propositional formula is an expression that consists of logical operators $\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow$ and a set of boolean variables. The truth values *true* and $false$ are assigned to the boolean variables in order to compute the result of an expression. We were already confronted with some small propositional formulas as part of the CTCs in a feature model. Each feature model can be automatically translated into a propositional formula [Bat05, Man02]. A feature is considered as a boolean variable and assigned the respective truth value (true = selected, false = not selected). The formula evaluates to true, if the selection of features is valid. Given the logical operators, every parent-child relationship can be translated into a propositional formula (cf. Table 2.1). A conjunction of all formulas represents the complete feature model [Bat05].

Relationship	Propositional Formula
Mandatory	$Parent \Leftrightarrow Child$
Optional	$Child \Rightarrow Parent$
Or	$Parent \Leftrightarrow (Child_1 \vee Child_2 \vee ... \vee Child_n)$
Alternative	$(Child_1 \Leftrightarrow (\neg Child_2 \wedge ... \wedge \neg Child_n \wedge Parent)) \wedge$
	$(Child_2 \Leftrightarrow (\neg Child_1 \wedge ... \wedge \neg Child_n \wedge Parent)) \wedge$
	$(Child_n \Leftrightarrow (\neg Child_1 \wedge ... \wedge \neg Child_{n-1} \wedge Parent))$

Table 2.1: Translation from feature model to propositional logic.

Delta Modeling. In order to create a seamless development process for SPLs, we need a variability modeling technique for the solution space as well. Delta Modeling (DM) is a general concept of integrating variability with atomic model transformations. Variability in DM is captured by a core model and a set of delta models. A core model corresponds to a valid variant, and in many cases, it is the smallest possible variant of the SPL. The delta models contain changes to the core model by additions, modifications and removals of model elements. An application of a valid set of delta models to the core leads to a new variant of the SPL. Fig. 2.4 depicts the principle of DM using the example of state machines. First, we create a core model of the product line. In this example, the core model contains four connected states including an initial and final state. Next, we can define the necessary model transformations in the delta models. For instance, the delta model to generate *Variant 1* based on the core model contains the removal of the transition from *S2* to *S1*. We can define the delta operations to arrive at *Variant 3* in a similar way. In addition, DM gives us the instruments to also generate *Variant 3* based on *Variant 1*, if we define the necessary delta operations. This process can be continued to derive the other variants displayed in Fig. 2.4 as well.

Furthermore, we can attach an application condition to every delta model enabling the selection of deltas based on a concrete feature combination. Therefore, application conditions are defined in terms of features from the feature model. In combination with an application order for delta models, we can also compose multiple deltas to derive a concrete variant [Sch10]. The main characteristics of DM are as follows [Sch10]:

- The DM concept is independent of a concrete implementation or modeling language. We can instantiate it for any language, e.g. Java or the Unified Modeling Language, by defining the respective delta operations.
- DM allows a modular as well as incremental development process.
- Variability is expressed by the same concepts across all models of the SPL.
- Feature combinations can be explicitly captured by the attached application conditions.

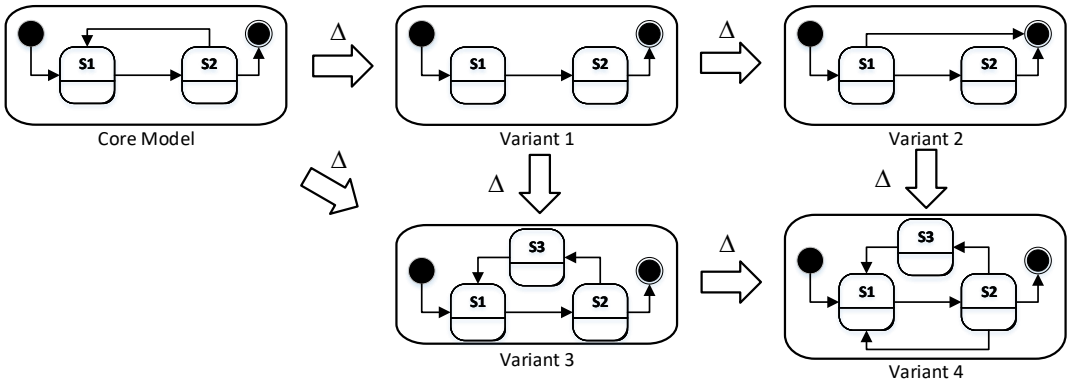


Figure 2.4: Principle of Delta Modeling [Sch10].

A formal definition of DM and its semantics is given in [Sch10]. Furthermore, DM is not the first and not the only available variability modeling concept in the literature. A comparison of several approaches is given in the related work (cf. Section 4.4).

Software Product Line Analysis Strategies. The analysis of a software system is a crucial task to ensure its correctness and reliability. However, the large variant space in SPLs threatens the application of traditional analysis strategies such as type checking, theorem proving or performance analysis. The generation of all possible variants already is infeasible in many cases due to the potentially exponential growth of valid feature combinations. As a result, researchers started to exploit the very nature of SPLs and developed analysis strategies that consider the variability information available in the problem and solution space models [TAK+14, CE05]. In the following, three analysis strategies are described that we encounter throughout the course of thesis.

The naive approach is a product-based analysis in which we generate (all) products of the SPL and analyze them in isolation. A typical optimization strategy is to compute a sample set containing a smaller number of products based on a coverage criteria. This subset is then used to make statements about properties for the complete SPL. However, the core strategy remains a product-based analysis, which is defined as follows:

Definition 2.6: Product-based Analysis

"The analysis of an SPL is *product-based*, if it operates only on generated products or models." [TAK+14]

Its main advantage lies within the applicability of existing tools to analyze individual products. Additionally, the analysis of all products in this manner provides a sound and complete result for a considered property. A product-based approach typically serves as baseline to compare other strategies, e.g., in terms of efficiency (cf. Chapter 5). Of course, the main disadvantage is the inefficiency due to redundant computations in this strategy [TAK+14].

Family-based analysis strategies constitute the second type. The general idea is to analyze the variability model and domain artifacts together by creating a single large model and derive the intended properties afterwards. The definition is as follows:

Definition 2.7: Family-based Analysis

"The analysis of an SPL is *family-based*, if it operates only on domain artifacts and incorporates the knowledge about valid feature combinations." [TAK+14]

Thus, family-based strategies can avoid redundant computations and it is not necessary to generate individual products. We can typically observe an increased

efficiency, e.g., in terms of computation time. The main drawback is that changes in the variability model, i.e., the addition of a new feature, forces us to create a completely new model and execute the analysis again. In addition, the size of the analysis problem may exceed physical limitations such as memory capacities as we consider all artifacts of the SPL at once [TAK+14].

Further analysis strategies are defined in the literature such as feature-based approaches, which consider only domain artifacts related to a specific feature. Finally, a combination of multiple approaches is possible. For instance, a family-product-based analysis is defined as:

Definition 2.8: Family-Product-based Analysis

"The analysis of an SPL is *family-product-based*, if it consists of a family-based analysis followed by a product-based analysis and the analysis effort of the family-based analysis is reused in the product-based analysis." [TAK+14]

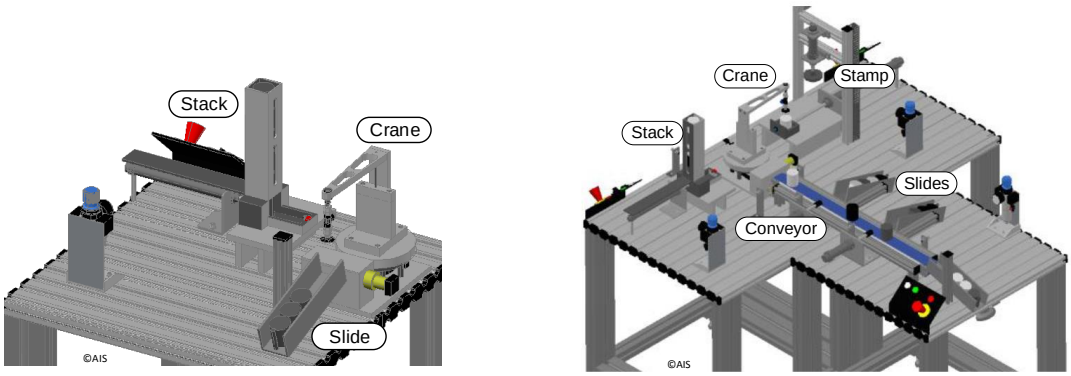
In Chapter 5, we develop a family-product-based analysis strategy for performance properties and compare it to a product-based approach. Chapter 4 contains two product-based approaches used to ensure the consistency in solution space models.

2.2 The Pick and Place Unit as Running Example

The Institute of Automation and Information Systems (AIS) at the TU Munich has developed a bench-scale automation system called the Pick and Place Unit (PPU) [VHLFF14, LFVH13]. The development of a case study usually includes significant resources, time- and money-wise, and this is especially the case in the domain of plant and machine engineering. An automation system is comprised of both software and expensive hardware components, while its evaluation capabilities are limited compared to pure software systems due to the nature of such an automation system [VHLFF14]. The estimated cost of the PPU hardware is about 100.000 Euros. The main purpose of the PPU is to study variability aspects in the automation domain. For instance, customer A may require faster robots than costumer B, which means that the hardware setup as well as the software have to be adapted.

Design of the Pick and Place Unit. Fifteen different variants of the PPU are defined and documented with, e.g., structural and behavioral models, CAD, simulation, and control code for programmable logic controllers [VHLFF14]. All variants of the PPU share a common set of features fulfilling Def. 2.2 for an SPL. These features are as follows:

- Stack: Functioning as a workpiece input storage.
- Slide: Functioning as a workpiece output storage.
- Crane: Transporting workpieces between the input and output storage.



(a) PPU - Basic Variant

(b) PPU - Full Variant

Figure 2.5: CAD models of the PPU based on [VHLFF14].

While the stack and crane are software-intensive components, the slide does not contain any software itself, since it is just a mechanical ramp. Fig. 2.5a depicts the CAD model of these core features that simultaneously compose the smallest possible variant of the PPU. Each of the available variants provides an optimization of the system or adds some new functionality. Fig. 2.5b shows the maximum variant of the PPU containing all features. Two larger features were added with a stamping module and a conveyor to improve the output storage. Of course, several smaller changes to software and hardware components were made in order to reach this final variant that are not directly visible in this high-level CAD model. In theory, it is possible to create many more variants through combinatorial combination of individual features (cf. Section 3.1). However, only the 15 variants are reasonable and therefore well-documented by the developers. Throughout this thesis, we consider the following three variants in more detail:

Basic: Is the minimal system configuration consisting of three components, i.e., stack, crane and slide (cf. Fig. 2.5a). The workpieces are transported from the stack to the slide by the crane. This process is repeated until no more workpieces are present.

Stamp: In the next more complex variant, the PPU can distinguish between two different types of workpieces (metallic or plastic). While black-colored plastic workpieces are treated as in the **Basic** variant, metallic pieces take a different route through the system. They are transported by the crane to the new stamping component. After the stamp process is finished, these pieces are also transported to the slide. The process is repeated until no more workpieces are available at the stack.

Optimized: This variant is identical to the previous one on the hardware level. The crane implementation is optimized as the crane no longer waits at the stamp for the stamping process to be finished. Instead, the crane moves back to the stack to pick up the next plastic workpiece (if present) and transports it to the slide. Afterwards, the crane returns to the stamp and picks up the stamped metallic workpiece and also transports it to the slide.

A detailed description of the specific hardware changes including sensors and switches as well as implementation artifacts for all variants can be found in the technical report, which is provided by the AIS chair [VHLFF14]. Whenever applicable throughout this thesis, we use the PPU to explain new aspects relevant for the specific contributions. However, the PPU is still a small SPL and not all proposed contributions are visible if we consider only our running example. This is especially the case considering later evaluations as the contributions are intended for industrial-size SPLs.

3 Interdisciplinary Variability Modeling in the Problem Space

This chapter shares material with work published in [KAT16], [AKTS16] and [KATS17].

Contribution

We adapt an artificial intelligence algorithm to explain defects that can appear in a feature model. The explanations are presented in a user-friendly way, and their computation has almost no impact on the performance of the application. Given the possibility of multiple interrelated models, we also identify and explain hidden dependencies by the same means. The applicability is shown in a two-fold evaluation using our running example and industrial-size SPLs.

Variability modeling is a core task during SPL development. The resulting model is a cornerstone in each SPL serving as foundation for communication purposes with the customer as well as the derivation of complete product variants (cf. Section 2.1.2). An established approach to model variability in SPLs are feature models [PBvdL05, CE05, BSRC10]. A feature represents a commonality or difference in the SPL. Industrial-size SPLs tend to contain thousands of such features and relationships between them making the development and maintenance increasingly more difficult. Developers can easily introduce defects in the feature model during its evolution [MD08]. This problem is even further intensified by considering multiple interrelated feature models for different engineering disciplines or purposes. Each feature model is maintained by several developers. Supporting both development and maintenance in this topic is part of the automated analysis of feature models and one of the main challenges in SPLs [BFGR13, BSRC10].

In this chapter, we propose an adapted artificial intelligence algorithm to generate user-friendly explanations for possible defects in a feature model based on its underlying propositional logic representation. As a preliminary step, we are also able to identify hidden dependencies if we encounter multiple interrelated feature models. The algorithm is able to explain these dependencies in a similar way.

We first lay the necessary foundation for this contribution with background on defects and automated analyses in feature models. The preliminaries are followed by the two main concepts of identifying hidden dependencies and generating explanations for the defects as important aspects in the automated analysis of feature

models. Afterwards, we present an evaluation in order to emphasize the qualitative and quantitative results of both concepts. Next, we compare our concept with already existing ideas in the literature. Finally, the chapter is concluded by a summary and discussion of points left for future work.

3.1 Preliminaries

An introduction to feature modeling and its concepts was already given in the background (cf. Section 2.1.3). Thus, we can continue with the description of defects in feature models.

3.1.1 Defects in Feature Models

The development and maintenance of a feature model is always prone to the introduction of defects, especially with a rising number of features. However, such problems can only occur by defining additional relationships using CTCs or an equivalent approach. The general parent-child relations are error-free, if we consider the four common semantics described previously (cf. Section 2.1.3). Von der Maßen et al. classify possible defects into three major categories [vdML04]:

1. Inconsistencies: A feature model contains an inconsistency, if a contradiction between modeled relationships is present ultimately leading to an inconsistent product derivation. Inconsistencies pose severe issues that can occur on domain and product configuration level. A domain inconsistency reflects an erroneous captured domain, e.g., a mutual exclusion between two core features. A product configuration inconsistency results from invalid configurations, e.g., a core feature has not been selected for a product.
2. Anomalies: A feature model contains an anomaly, if at least one configuration cannot be derived although it is valid. Anomalies are categorized as medium issues, e.g., a core feature that implies an optional feature resulting in the optional feature to be a core feature as well. Thus, a variant without the optional feature is not possible.
3. Redundancies: A feature model contains redundancy, if at least one relationship is modeled multiple times. Redundancy is a double-edged sword. It decreases maintainability, since all appearances of the relation have to be changed in case of evolution. Yet, sometimes it can increase the comprehensibility of a feature model for developers. Thus, it is classified as a light issue, e.g., an optional feature that implies a core feature which is already in all variants.

In the following part, we introduce specific defects that can occur in a feature model and classify them according to the described categories. The PPU does not contain all identified defects which is why we use a different example. Fig. 3.1 shows a significantly simplified feature model of a car. A car must have a *Carbody* and a

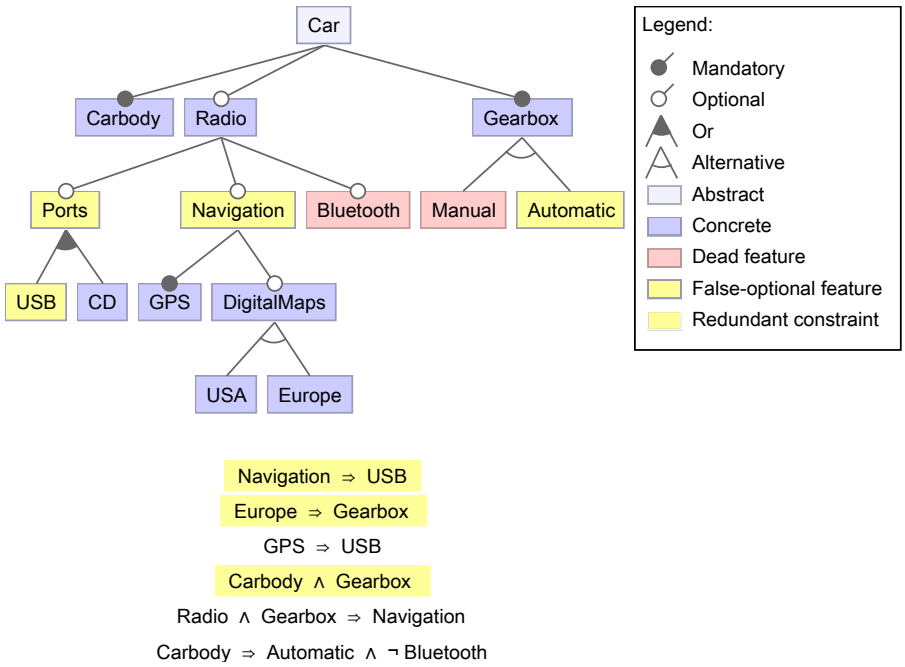


Figure 3.1: Defects in a feature model [KAT16].

Gearbox. One can choose between *Manual* and *Automatic* gearboxes. In addition, we can select a *Radio* with different subfeatures, namely *Ports*, *Navigation*, and *Bluetooth*, further extending the functionality. *Navigation* automatically includes a *GPS* system and may include maps for either *Europe* or *USA*. Music can be played via *USB*, *CD*, or both options. Finally, six CTCs express further relationships in the feature model [KAT16].

Void Feature Models. A *void* feature model is categorized as an inconsistency, since not a single variant can be derived from the SPL. It is the most severe issue, which we can encounter as part of the feature modeling process [vdML04].

Example 3.1: Void Feature Model

For example in Fig. 3.1, adding the constraint $\neg(Carbody \wedge Gearbox)$ would result in a void feature model. Both features represent core features and a mutual exclusion between them adds a contradiction to the feature model making the derivation of a product impossible [KAT16].

Dead Features. Features are regarded as dead, if they can never be selected in any variant of the product line. Hence, the feature has no effect at all. A dead feature is classified as an anomaly, since only a subset of variants is not derivable [vdML04].

Example 3.2: Dead Features

The last CTC in Fig. 3.1 makes two features dead with *Bluetooth* and *Manual*. Due to the mandatory nature of *Carbody*, we must select an *Automatic* transmission. *Automatic* itself is part of an alternative group and therefore excluding all other features in this group, here only *Manual*. In addition, the CTC forbids *Bluetooth* to be part of the variant making it also a dead feature [KAT16].

False-Optional Features. A feature is defined as false-optional, if the selection of its parent makes the feature itself selected as well, although it is defined as optional and not mandatory. False-optional features are also considered as an anomaly [vdML04].

Example 3.3: False-Optional Features

In Fig. 3.1, we already know that *Maunal* is a dead feature. Thus, the *Automatic* gearbox is always selected making it false-optional. The selection of a *Radio* implies the child feature *Navigation* as well producing a chain of effects, since *Navigation* also requires *USB* and therefore its parent *Ports* [KAT16].

Redundant Constraints. A CTC is redundant if its removal does not change the validity of configurations. There are multiple cases leading to a redundant CTC, e.g., the implication of a mandatory feature, mutual exclusion of alternative features, multiple implications and exlcusions or a transitive chain of implications. As the name already suggests, these defects fall into the third category of redundancy and are light issues, since no variant is actually affected [vdML04].

Example 3.4: Redundant Constraints

Consider the CTC $Carbody \wedge Gearbox$ in Fig. 3.1. It is redundant simply because both features are already mandatory and part of all product variants. A similar example is the CTC $Europe \Rightarrow Gearbox$, because requiring a mandatory feature is information that is already available in the feature tree. Redundancy can also be caused by the cooperation of numerous CTCs as in the case of $Navigation \Rightarrow USB$. By selecting the *Navigation*, it is already implied that the car has a *USB* port. Hence, it is not necessary to imply *USB* again with the *GPS* feature, which is automatically selected after *Navigation* because its mandatory. We could safely remove one CTC.

$$
\begin{aligned}
void(FM) &:= \neg SAT(FM) \\
dead(f) &:= \neg SAT(FM \wedge f) \\
falseOpt(f_{opt}) &:= TAUT(FM \wedge f_p(f_{opt}) \Rightarrow f_{opt}) \\
redundant(c) &:= TAUT(FM' \Leftrightarrow FM' \wedge c) \\
with\ TAUT(x) &:= \neg SAT(\neg x)
\end{aligned}
$$

Table 3.1: Defect detection with a SAT solver. FM = feature model, f = feature of interest, f_{opt} = optional feature, f_p = parent of feature f_{opt}, c = cross-tree constraint, $FM = FM' \wedge c$, and x = propositional formula [KAT16].

3.1.2 Automated Analysis of Feature Models

During the last section, the detection and explanation of defects was executed manually using expert knowledge. However, both methods get increasingly more difficult considering large-scale SPLs. Thus, we require analysis techniques to automatically identify and explain the defects. Since the invention of feature models in the year 1990, researchers have proposed several analysis techniques tailored to SPLs. The detection and explanation of defects pose two of challenges in SPLs that can be solved automatically [KCH+90, TAK+14, BSRC10].

Detection of Defects. Propositional formulas are an efficient representation of feature models in order to perform automated analyses techniques and each formula can be transformed into *Conjunctive Normal Form* (CNF). Given this conjunction of clauses, we can determine if the formula is satisfiable, thus answering the question whether we can derive a valid variant of the SPL in which case the feature model is not void. This check can be automated using a satisfiability (SAT) solver [BSRC10, Bat05]. Although SAT solving is NP-complete, a tremendous amount of research was conducted in order to make SAT solving and its tools much more efficient [Mar09].

Table 3.1 refers to SAT solver calls for the detection of defects. In case of a void feature model, the SAT solver cannot find any truth value assignment that gives us a satisfied formula. It is similar for the identification of a dead feature. A SAT solver tries to determine a truth value assignment in which a certain feature is selected. If no such valid assignment is found, the feature is marked as dead. For false-optional features, the solver is not able to find a solution in which the parent feature is selected while the child remains deselected. Thus, the child must always be selected and is marked as false-optional. The respective call in Table 3.1 is always true and therefore a tautology. Redundancy is identified by checking whether two feature models are equivalent. One feature model contains the considered constraint,

while it is not present in the other model. The derived configurations or truth value assignments satisfying the formula must be identical in order to identify the constraint as redundant. The automated detection is a prerequisite to actually explain defects. We rely upon an existing framework for feature modeling, which already includes detection of defects using a SAT solver [KTS+09].

The detection of defects is not restricted to the domain of SAT solving. Another strategy relies on *binary decision diagrams* (BDD). The propositional formula of the feature model is transformed into a BDD instead of a CNF. Again, we can check if a certain truth value assignment is satisfiable. In addition the paths in a BDD represent all possible product configurations of the SPL revealing all variants. However, the size of a BDD heavily depends on the ordering of variables and finding the best variable ordering is also NP-complete. A combination of several solvers may lead to a more efficient solution by taking individual advantages of the strategies into account for a specific problem, e.g., defect detection [BSTRC07].

Explanation of Defects. The generation of an explanation for a defect is the second automated analysis considered in the literature [BSRC10, BFGR13, LSW15]. An explanation should inform developers about the features and relationships leading to a defect [BSRC10]. For example, consider the following simple feature model:

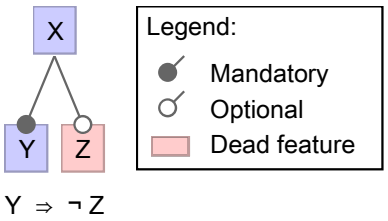


Figure 3.2: Simple feature model containing a dead feature.

In this case, it is obvious why the feature Z is detected as dead. Three available explanation approaches in the literature give us the following feedback:

- **Approach 1** [Bat05]:
  X has a contradiction Z because set by user
  not Y because (Y) implies (not Z)
  not X because (X iff Y)
  not X because it is root of grammar
  But X

- **Approach 2** [TBD+08]:
  1: Dead Feature: Z (2 explanations) -> [to-Y-rel], [CTC-1]

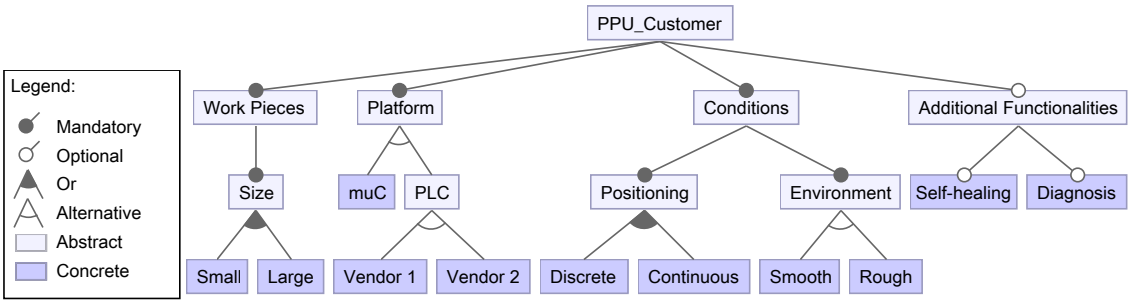


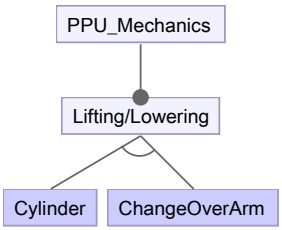
Figure 3.3: Feature diagram of the PPU: Customer Viewpoint [FLVH15, LFVH13, AKTS16, KATS17].

- **Approach 3** [RGMS14]:
  Analyzing the defects...
  Defect:.....DEAD_FEATURE
  Feature:.....Z
  Causes: FULL MANDATORY FEATURE EXCLUDES AN OPTIONAL FEATURE
  Explanation: Optional feature Z is dead because it is excluded by the full mandatory feature Y with the dependency CTC1

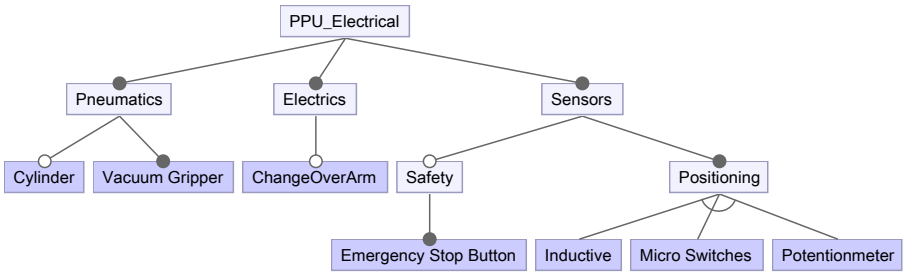
The generated explanations range from just presenting the responsible CTC as well as the parent-child relationship between X and Y to full natural language expressions. A detailed description of the underlying mechanics is omitted at this point and given in the related work (cf. Section 3.5).

3.2 Interrelated Feature Models

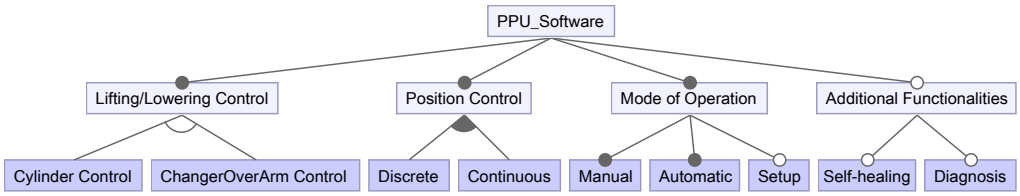
A typical feature model of the industrial domain has several thousand features and CTCs making maintenance a tedious and error-prone task. Multiple developers from different domains further contribute to this problem. One possibility to counteract such negative aspects in large-scale SPLs is the usage of multiple feature models that are dedicated to separate purposes, scopes or levels of granularity. The purpose of a feature model is described as its considered space. Features can exist in the problem-, solution- and configuration space. Problem space features rather refer to a system specification in terms of requirements identified during the domain analysis. Solution space features refer to concrete system artifacts for the development and configuration space features deal with product derivation aspects. In classical feature modeling, the scope is identical to the complete SPL. We have one model containing all necessary features. However, even a small SPL as the PPU can easily be divided into several reasonable and separate scopes (cf. Fig. 3.4 to get a first idea). As a



(a) Feature diagram of the PPU: Mechanical Viewpoint



(b) Feature diagram of the PPU: Electrical Viewpoint



(c) Feature diagram of the PPU: Software Viewpoint

Figure 3.4: Engineering feature diagrams of the PPU [FLVH15, LFVH13, KATS17, AKTS16].

result, the complexity for individual developers is reduced, since they only have to focus on a subset of all features. Developers must always decide the desired level of granularity for their feature model. Features can be high-level components as the crane or just a small sensor in case of the PPU [LEGP15].

For instance, Fig. 3.3 is solely comprised of features in the problem space, since it is meant for a potential customer and visualizes the requirements identified for the PPU. The PPU can process up to two different types of workpieces simultaneously with size *Small* and *Large*. The operating environment can either be *Smooth* or *Rough*, but not both at the same time. The additional functionalities of *Self-healing* and *Diagnosis* are optional. The customer feature diagram contains only 20 features, but we are already able to generate 270 valid configurations. Thus, 270 potential PPUs although only 15 variants are actually described and implemented (cf. Section 2.2).

However, the development and maintenance of the PPU involves multiple domains with software, mechanical and electrical engineering that are not sufficiently represented in the previously described feature model. The model in Fig. 3.3 is solely for configuration purposes by a potential customer or plant manufacturer. As a result, three additional engineering feature models are available for the PPU covering the different domains [FLVH15, LFVH13]. The individual models are depicted in Fig. 3.4. It is not surprising that some features are similar to the customer feature model such as the *Positioning* (present in the electrical and software model) or *Additional Functionalities* that are also visible in the software feature model. However, some features are only present in the engineering models, e.g., *Safety*. Fig. 3.4a consists of two alternative features describing different mechanical mechanisms to lift workpieces. These features are also reflected in Fig. 3.4b as child features of *Pneumatics* and *Electrics*. The PPU can be equipped with several sensors, namely *Inductive*, *Micro Switches* or *Potentiometer*, to ascertain the positions of workpieces and the crane. For the PPU, an *Emergency Stop Button* is not mandatory. Depending on the selected sensors, the positioning can work in a discrete or continuous manner (cf. Fig. 3.4c). The PPU can always be controlled manually or operate fully automatic. Finally, the PPU can have a diagnosis function and heals itself if a problem is encountered. It is not relevant for our work, to what extent self-healing is possible [AKTS16, KATS17]. The goal of separate feature models is to reduce the complexity for the developers and let them focus on important assets for their domain [STSS13, ACLF11a].

3.2.1 Combining Interrelated Feature Models

Several feature models alone do not provide an ultimate solution to reduce complexity, since such an approach is missing a very important aspect. The individual feature models have to be connected to each other in order to be able to fully model the SPL. Hence, we need a mechanism to express dependencies between the separate feature models. Again, CTCs are a viable solution for this problem [AKTS16, KATS17]. For instance, the CTC $Smooth \Rightarrow (Micro\ Switches \vee Inductive \vee Potentiometer)$ specifies that by selecting the feature *Smooth* in the customer feature model of the PPU, we also have to select one of the positioning sensors in the electrical feature model of the PPU.

The developers of the PPU models proposed a different approach using a mapping matrix between the customer feature model and the engineering models to express such global constraints [FLVH15, LFVH13]. For instance, a customer can select the small workpiece type resulting in the selection of the features *ChangeOverArm* in the mechanics, *Vacuum Gripper* and *ChangeOverArm* in the electrics and *ChangeOverArmControl* in the software. Several of these dependencies are defined

				Developer's point of view					
				Mechanical	Electrical			Software	
				Lifting/Lowering	Pneumatics	Electrics	Sensors	Lifting/Lowering Control	Position Control
Customer's point of view	Work pieces	Size	Small	ChangeOverArm	Vacuum Gripper	Change OverArm		ChangeOverArm Control	
			Large	ChangeOverArm / Cylinder	(Cylinder) Vacuum Gripper			(Cylinder Control)	
	Conditions	Positioning	Discrete				Micro S. / Induct. / Potentio.		Discrete
			Continuous				Potentio-meter		Continuous
		irronment	Smooth				Micro S. / Induct. / Potentio.		

Figure 3.5: Extract from the mapping matrix [FLVH15].

by the developers in [FLVH15]. Hence, the mapping matrix sets the customer's model into context of the engineering models. Fig. 3.5 shows an extract of the original matrix. The columns 1-4 (left-side) represent features from the customer feature model. The three top-level rows belong to the engineering models. The matrix is interpreted as follows: Selecting the small workpiece size in the customer model requires changeover arm in the mechanics, vacuum gripper in the electrics and so on. The connection of the individual engineering models to each other is missing in the mapping. Additionally, the definition is rather informal and the matrix does not scale very well. Thus, automated analyses for defect detection and explanation are difficult based on the mapping matrix making a representation as CTCs a more viable option [AKTS16, KATS17].

In the literature, the modularization of an SPL into multiple feature models and their relations is clustered under the emerging topic of multi software product lines [HGR12]. A detailed description of existing concepts in this topic is presented in the related work part (cf. Section 3.5). In order to follow the proposed contribution, it is sufficient to know that we can combine multiple feature models using CTCs and feature models themselves can be translated to propositional logic as well.

Implicit Constraints. The connection of multiple feature models with CTCs or a mapping matrix to depict global relationships may lead to hidden dependencies in the individual feature models. Developers often focus on just one feature model and are not aware of all the global constraints. However, they should be informed of the resulting dependencies for their feature model, since they can easily introduce defects otherwise [LEGP15, KST+16]. For the remainder of this thesis, we call such hidden dependencies *implicit constraints*.

(a) Hidden implication

(b) Hidden exclusion

Figure 3.6: Two exemplary implicit constraints [AKTS16, Ana16].

Example 3.5: Implicit Constraints

Fig. 3.6 shows four feature models. The left side describes a cycle of implications between two feature models due to two global constraints (cf. Fig. 3.6a). Feature C implies the feature G and in return G also implies D resulting in an implicit constraint in the *Feature Model 1*, namely $C \Rightarrow D$. The right side depicts another appearance of an implicit constraint (cf. Fig. 3.6b). Again two global CTCs are defined with feature C implies G and D requires H. However, G and H are part of an alternative relationship and due to the CTCs the optional features C and D are also mutual exclusive with $\neg(C \wedge D)$ [AKTS16, Ana16].

An implicit constraint is a projection of globally defined relationships across multiple feature models to a concrete relationship relevant in a specific feature model. Thus, implicit constraints are always redundant as their contained information is identical to the global relationships from which they originate [AKTS16]. We can also apply this statement to a single feature model as explained in the next section.

3.2.2 Making Hidden Dependencies Visible

The PPU consists of four feature models of which each depicts a different viewpoint. In addition, we are aware of some relationships from the customer model to the individual engineering models. A feature model representing the full product line is called a *complete* feature model. This includes separate feature models for different domains, since they can be merged into one large model under a new root feature. In case of the PPU, we can integrate the four models at hand into a complete model. Most of the automated analysis techniques operate on a single large model [BSRC10].

A *partial* feature model is an arbitrary submodel in this complete model. It can either be a model representing one domain, e.g., the customer model for the PPU

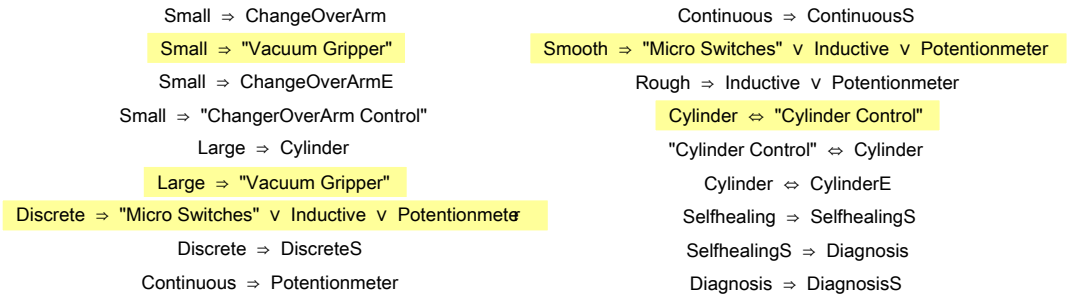


Figure 3.7: CTCs based on the mapping matrix. Highlighted constraints are redundant [FLVH15, AKTS16, KATS17].

or a submodel in a feature model, e.g., the *Work Pieces* with all of its subfeatures represents a submodel. The idea of a partial feature model is to reduce the visible complexity for the developer [AKTS16].

Implicit constraints occur if we only consider such partial feature models. They always provide redundancy considering the complete feature model, since the information is already available in the CTCs of the complete feature model, which we refer to as global constraints. Showing globally defined CTCs does not help in a partial feature model if the features are unknown and the number can easily add up to several thousands. Stakeholders need to focus on constraints relevant for the partial feature model that they observe at the moment. In addition, it is mandatory that developers are aware of such dependencies in order to prevent the introduction of inconsistencies or errors during feature model maintenance and development [AKTS16].

In order to reason about the implicit constraints of the PPU, it is first necessary to convert the mapping matrix into a representation that is efficient to analyze such as propositional formulas. Hence, we translated the matrix into CTCs. The complete list of constraints is depicted in Fig. 3.7. They represent our global constraints that must be valid the complete feature model. Highlighted constraints are redundant and provide no additional information, thus it would be possible to remove them without introducing defects or false configuration possibilities [KAT16, FS10]. We validated the correctness of **all** CTCs with the actual developers of the PPU. As previously mentioned, 15 variants of the PPU are described in detail. The actual feature models permit a significantly larger number of possible variants with 5184, since the mapping matrix is not restrictive enough [AKTS16]. If the global constraints of the complete feature model are already defined as CTCs, we can skip this conversion step.

Detection of Implicit Constraints. To assure the completeness of partial models, the automated detection of hidden dependencies has become an important field of research [LEGP15, KST+16, SKT+16, ACLF11b]. For our approach, the detection is a necessary prerequisite to actually explain why implicit constraints occur [AKTS16].

The first challenge is the creation of a partial model based on a complete one, while preserving all dependencies. The elimination of features must not change dependencies between features in the submodel. A state-of-the-art approach for this problem is feature model slicing [ACLF11b]. It removes features while preserving all existing dependencies. Feature model slicing is typically used for the removal of abstract features or the decomposition and evolution of a feature model [TKES11, ACLF11b]. We benefit from an efficient slicing algorithm that has already been successfully applied in practice [KST+16, SKT+16]. Inputs to this slicing algorithm consist of a complete feature model in CNF and a subset of features that are not part of the desired partial feature model. After performing the slicing, the algorithm returns a partial feature model in CNF without the specified set of features while maintaining all dependencies between features in the partial model. The foundation of the slicing algorithm is based on *logical resolution*. The idea is to construct a new CTC, namely *resolvent* r_s, that represents a dependency in the partial feature model. Given two clauses c_1, c_2 and a literal $l \in c_1 \wedge \neg l \in c_2$, we can derive a resolvent using the resolution rule by combining both clauses and removing the literal l such as $r_s = (c_1 \cup c_2) \setminus \{l, \neg l\}$. The resolvent represents a transitive relationship between the two clauses and is considered as an implicit constraint. A brief example of the concept of logical resolution is given in the following and a more detailed explanation of the slicing algorithm is available in [KST+16].

Example 3.6: Logical Resolution

Remove B from the formula $(A \Rightarrow B) \wedge (B \Rightarrow (C \wedge D))$ with logical resolution:

1. Create CNF: $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D)$
2. Derive Resolvents:
   - $r_{s1} = ((\neg A \vee B) \cup (\neg B \vee C)) \setminus \{B, \neg B\} \Rightarrow (\neg A \vee C)$
   - $r_{s2} = ((\neg A \vee B) \cup (\neg B \vee D) \setminus \{B, \neg B\} \Rightarrow (\neg A \vee D)$
3. Add to input formula: $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D) \wedge (\neg A \vee C) \wedge (\neg A \vee D)$
4. Remove clauses used for resolution: $A \Rightarrow (C \wedge D)$

We reuse this feature model slicing algorithm in our approach. A specific feature, that acts as the root of the partial feature model, is selected. The slicing algorithm then removes all features that are not part of the subtree of the new root feature. Given the PPU, the process works as follows: First, the four feature models are merged into one large model. Therefore, we create a new root feature, e.g., *PPU* and the old root features, namely *PPU_Customer*, *PPU_Mechanics*, *PPU_Electrics* and *PPU_Software*, act as children of *PPU*. Second, we can select any feature in the complete model, e.g., one of the old root features, and the output of the slicing algorithm is the desired partial feature model with all dependencies.

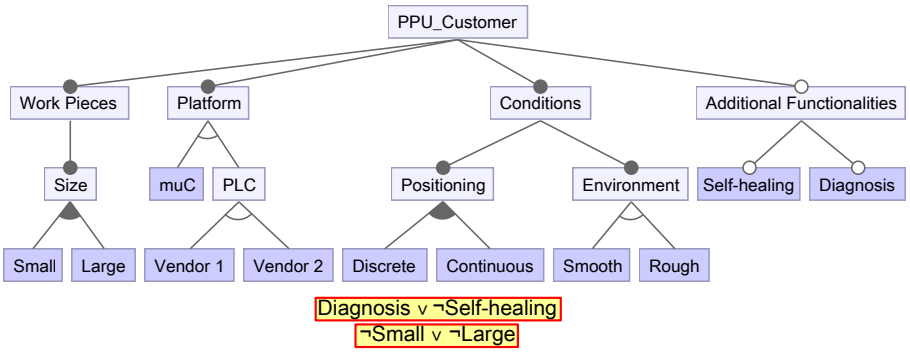


Figure 3.8: Customer model with implicit constraints [KATS17].

Highlighting Implicit Constraints. The sliced feature model might contain CTCs from the complete feature model. This is the case if a global CTC is defined solely with features of the partial/sliced model, e.g., the constraint only includes features from the customer model such as $Small \Rightarrow Discrete$. In order to detect implicit CTCs, it is necessary to differentiate between old global constraints and new implicit constraints by comparing CTCs of both feature models. Each CTC in the partial feature model that is not present in the global CTCs automatically is an implicit constraint considering this partial model. We decided to explicitly highlight implicit constraints with a red frame (cf. Fig. 3.8). Additionally, implicit constraints are always marked as redundant indicated by the yellow background (cf. Fig. 3.7 and Fig. 3.1). Global constraints in a partial feature model are not highlighted except they were also marked as redundant before the slicing process.

For the customer model, we compute two implicit constraints with $(Diagnosis \vee \neg Self\text{-}healing)$ and $(\neg Small \vee \neg Large)$ (cf. Fig. 3.8). As for the PPU, we have already shown the global constraints in Fig. 3.7. Since the individual feature models of the PPU have no constraints at all, it is obvious that no global constraint occurs after the slicing process for the customer model. The hidden dependencies are now visible to the developer. At this point, we are still missing the cause why these constraints occur and what the connection to the other feature model parts is.

3.3 An Algorithm for the Generation of Explanations

Explaining defects in feature models is a challenging automated analysis [BSRC10]. However, generating answers in form of why a certain defect has occurred during, e.g., feature model evolution, leads to a significantly simplified and faster repair process for these defects. In this section, we explain our analysis method for explaining defects in feature models beginning with the introduction of an adequate basic algorithm.

Algorithm Requirements. It is our motivation to support the developer such that a solid decision process is possible regardless of the defect type. This led to the following three requirements for our algorithm:

1. *Generic*: The algorithm should be *generic* to cover all defects mentioned previously.
2. *Efficient*: The algorithm should scale to large feature models with thousands of features and constraints.
3. *Informative*: Explanations should be user-friendly and as short as possible. Explanation parts, which have a high probability to cause a defect, should be emphasized.

The foundations of the explanation algorithm are based on previous work by Batory and can be found in a tool called GUIDSL which is part of the AHEAD tool suite [Bat05]. It successfully adapts basic ideas of the *Boolean Constraint Propagation* (BCP) algorithm to provide justifications for selected and deselected features [FK93]. In GUIDSL, a user can select a specific feature and obtains feedback why other features are not available anymore. However, explanations are just given in terms of propositional formulas and GUIDSL only operates during the configuration process (cf. Section 3.1.2, **Approach 1**). It provides the user with explanations concerning the configuration possibilities of a product line. BCP is a sound algorithm and used as an inference engine for a *Logic Truth Maintenance System* [Bat05, KAT16]. In the following, we explain the LTMS and BCP in more detail.

3.3.1 Background: Logic Truth Maintenance Systems

Truth maintenance (TM) is typically used for implementing an inference system in the domain of artificial intelligence. The core of a truth maintenance system (TMS) is an inference engine that deduces assumptions about variable values and maintains reasons for the deduced decisions. A TMS can be used to execute a range of activities such as propositional deduction, reasoning, and explanation capabilities [Doy79].

Thus, it is not surprising that quite a number of TMSs exists that are fundamentally different at their core. We have justification-, assumption- and logical-based TMSs in the literature [Rut94, DK86, MS83]. For the purpose of explaining defects, we require a TMS that can process any propositional formula, works with logical operations and comprehends semantics behind the logic, e.g., a variable cannot be represented by a positive and a negative assignment simultaneously [Cop04]. Justification- and assumption-based TMSs are restricted to horn clauses (i.e., clauses

of the form $A \wedge B \wedge ... \wedge Y \Rightarrow Z$) [FK93]. A logical-based TMS fulfills all the necessary requirements and is additionally already implemented in GUIDSL providing us with a first starting point. The *Logic Truth Maintenance System* (LTMS) is a propagation-based approach working with boolean constraints in order to derive an explanation. The basic idea of LTMS is depicted in Fig. 3.9 and we need the following logical specification [KAT16]:

- a set of boolean variables,
- a propositional formula composed of clauses that restrict these variables,
- premises, which are initial and permanent truth value assignments for the variables, and
- a set of truth value assumptions for the variables that may be revoked at some point.

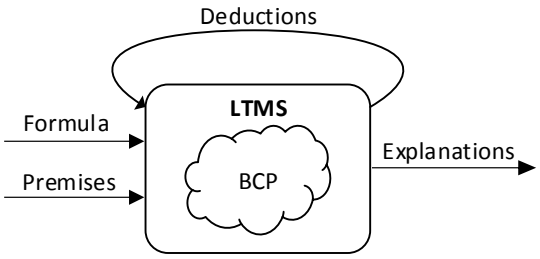


Figure 3.9: Logic Truth Maintenance System [KAT16, Ana16].

While an LTMS is a powerful approach capable of emulating a simple artificial intelligence, we focus on the ability to deduce inferences made from truth value assumptions and boolean variables. After an inference is computed, the LTMS saves the reason for the inference. We benefit from this process, since an LTMS is able to find violations in a propositional formula based on the premises and the inferred assumptions. For instance, the truth value of a dead feature has to be *false* in order to satisfy the feature model formula and by setting its truth value to *true* as premise, an LTMS will compute a violation at some point. We can reuse the inferred and stored reasons up to this point for the generation of an explanation [KAT16, Ana16].

The described logical specification can be directly mapped to our feature model domain with features represented by Boolean variables. Since each feature model can be translated to a propositional formula, we have clauses restricting the boolean variables. The premises depend on the considered defect which we plan to explain.

Boolean Constraint Propagation The fundamental core of an LTMS is its contained inference engine. Analog to the implementation in GUIDSL, we make use of boolean constraint propagation (BCP) as inference engine [Bat05]. The constraints

are represented by Boolean formulas and pose a special constraint satisfaction problem [Apt99]. Variables are connected by AND, OR, and NOT and we can reason about truth values given specific logical rules. Two rules are exemplarily shown below [KAT16]:

Example 3.7: Reasoning about Truth Values

1. $X \wedge Y = Z$: If Z = *true*, then X and Y must be true.
2. $X \vee Y = Z$: If X = *false* $\wedge$ Z = *true*, then Y must be true.

BCP is also called *Unit Resolution* and uses such rules in order to deduce inferences [DP60]. A three-valued logic serves as input to BCP with (true, false, unknown) as well as a propositional formula in CNF.

The basic principle of a BCP is to assign one of the following types to every clause in the CNF:

- *Satisfied*: at least one variable is true.
- *Violated*: all variables are false.
- *Unit-Open*: one variable is unknown while the remaining variables are false.
- *Non Unit-Open*: more than one variable is unknown, the remaining ones are all false.

Thus, we can observe that a unit-open clause can be satisfied by setting its unknown variable to true. A violated clause is equivalent to a contradiction necessary for the explanation of a defect. The BCP clause types are now demonstrated with a concrete example [KAT16]:

Example 3.8: Clause Types in BCP

Regarding the clause $\neg A \vee B \vee C$, the different types are demonstrated:

- If A is *false*, the clause is **satisfied**.
- If A is *true*, B is *false* and C is *false*, the clause is **violated**.
- If A is *true*, B is *false* and C is *unknown*, the clause is **unit-open**. C is derived as *true*.
- If A is *true* and B and C are *unknown*, the clause is **non unit-open**.

An overview of the BCP algorithm is given in Fig. 3.10. BCP is invoked with the CNF of the feature model and the initial truth value assignments of the premises. The selection of truth values for the premises is especially important, since we want

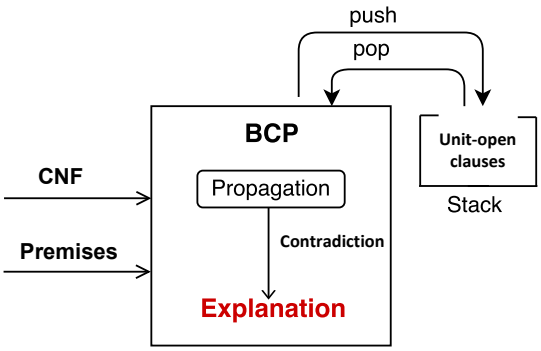


Figure 3.10: Overview of the BCP algorithm [KAT16, Ana16].

to force BCP to generate a contradiction. At first, BCP searches the CNF for unit-open clauses and pushes them on a stack. Next, the algorithm pops the last encountered unit-open clause from the stack and infers the truth value of the unknown variable. The result is stored and the CNF is searched again for new unit-open clauses that are pushed onto the stack. This process is repeated in an iterative fashion until a violation is detected during the constraint propagation and BCP terminates.

A three-tuple of $(conclusion, reason, \{antecedents\})$ is responsible for the information storage of each deduced truth value assignment. A *conclusion* represents an assigned truth value to a variable. The *reason* is the unit-open clause leading to the inferred value and the set of *antecedents* contains all the remaining variables in the considered clause whose truth values were already referenced and for which BCP also maintains such a three-tuple. We demonstrate the individual steps of the BCP algorithm as part of the following example [KAT16, Ana16]:

Example 3.9: BCP in Action

Given the formula of a feature model with $(A \Rightarrow B) \wedge (B \Rightarrow \neg A)$, which is transformed to a CNF: $(\neg A \vee B) \wedge (\neg B \vee \neg A)$. As depicted in Table 3.3.1, BCP is invoked with $A = true$ and maintains its reason as *premise*. Consequently, a premise does not have any *antecedents*. BCP pushes all unit-open clauses from the CNF to a stack. After examining $(\neg B \vee \neg A)$, BCP infers $B = false$, records its unit-open clause as a reason and refers to variable A as its *antecedent*. Next, BCP discovers the violated clause $(\neg A \vee B)$ and reports a contradiction.

ID	Conclusion	Reason	Antecedents	Stack
#1	A=1	premise		$(\neg B \vee \neg A), (\neg A \vee B)$
#2	B=0	$(\neg B \vee \neg A)$	#1	$(\neg A \vee B)$
#3		$(\neg A \vee B)$	#2	violated clause

Table 3.3.1 The general process of the BCP algorithm.

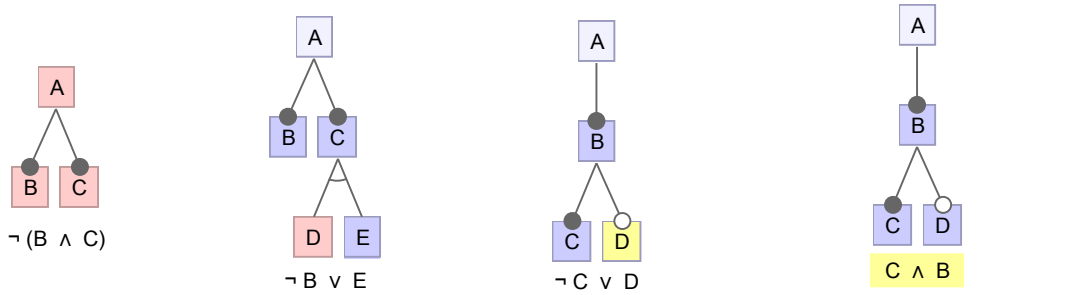
In GUIDSL, LTMS and BCP are used to support a user during the configuration process of an SPL by giving explanations in form of propositional logic stating why a specific feature cannot be selected (cf. Section 3.1.2, **Approach 1**) [Bat05]. In this thesis, we take LTMS and BCP on a different level by already supporting the actual development and maintenance of the feature model itself, explaining all described defects and presenting a comprehensible explanation to the developer.

3.3.2 Automated Explanation of Defects

BCP is able to explain defects in a feature model given its CNF and initial truth value assignments. The algorithm is generic, since we only have to adapt the input premises with respect to the considered defect. In the following, we give small use cases (cf. Fig. 3.11) for all considered defects serving as examples to demonstrate the applicability of the BCP algorithm.

Explaining Void Feature Models. The most severe defect is a void feature model. Consider the feature model depicted in Fig. 3.11a. The features B and C exclude each other, although both are core features. BCP requires the CNF of the feature model and a premise leading to a contradiction as input. In case of a void model, we can force the occurrence of a contradiction by assuming the root feature A as true, hence it is selected. Table 3.2 describes the step-wise execution of BCP. First, we create a CNF for the feature model. During the creation of the CNF, we additionally store information about the tracing of each variable to the feature model, since every variable belongs to a clause, which either originates from the feature tree topology or from a CTC. All variables except for the premise $A = 1$ are unbound at this point. BCP traverses the CNF and pushes all unit-open clauses on a stack. Afterwards, the clause at the top of the stack, i.e., $(\neg A \vee C)$, is removed and BCP infers the truth value of C satisfying the clause. All clauses containing C are updated with the inferred value. Next, the CNF is searched for new unit-open clauses and the process is repeated until a contradiction appears in the clause $(\neg A \vee B)$. Since the information of the tracing between all used clauses and the feature model is available, we can reuse it to create a comprehensible explanation. An explanation can be generated by reporting the reasons: first, take the violated clause into account and, second, traverse the reasons for conclusions backwards to the premises. Initial value assumptions do not need to be reported.

While a textual representation can be considered as a first step in terms of user-friendly explanations (cf. Table 3.2 bottom part), we ultimately prefer a visualization of the reasons directly visible in the feature model as depicted in Fig. 3.12. However, both explanation types are identical and contain the same information.



(a) Void Model (b) Dead Feature (c) False-optional Feature (d) Redundant Constraint

Figure 3.11: Examples used for the application of BCP [KAT16].

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg B \vee \neg C)$

ID	Con.	Reason	AC	Stack
#1	A=1	premise		$(\neg A \vee C), (\neg A \vee B)$
#2	C=1	$(\neg A \vee C)$	#1	$(\neg B \vee \neg C), (\neg A \vee B)$
#3	B=0	$(\neg B \vee \neg C)$	#2	$(\neg A \vee B)$
#4		$(\neg A \vee B)$	#3	violated clause

Explanation: *Feature Model is void, because: B is a mandatory child of A (violated clause), $\neg(B \wedge C)$ is a constraint (#3), C is mandatory child of A (#2).*

Table 3.2: Explaining the void feature model.

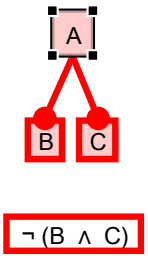


Figure 3.12: Result

Explaining Dead Features. An exemplary application to explain a dead feature is demonstrated with the feature model illustrated in Fig. 3.11b. An alternative feature E is implied by a core feature B which results in D to be dead. The respective BCP process including a resulting explanation is depicted in Table 3.3. The premise D = *true* is propagated. As D is initially bound, it does not have any antecedents and all other variables are assumed to be unknown. The last occurred unit-open clause $(\neg D \vee \neg E)$ is removed from the stack and examined. Since D is bound as *true* and the clause has to be satisfied, variable E is concluded to be *false*. The algorithm continues until a value is set resulting in a violated clause of the CNF, which is the case for the clause (A). A is the root feature and set to *false* in the fourth iteration of BCP resulting in a contradiction. The stack in iteration #5 is omitted at this point, but instead we present the violated clause of the CNF [KAT16, Ana16]. For the dead feature D, the explanation is shown on a textual level in Table 3.3 and graphically in Fig. 3.13.

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg C \vee D \vee E) \wedge (\neg D \vee C) \wedge (\neg E \vee C) \wedge (\neg D \vee \neg E) \wedge (\neg B \vee E)$

ID	Con.	Reason	AC	Stack
#1	D=1	premise		$(\neg D \vee \neg E), (\neg D \vee C)$
#2	E=0	$(\neg D \vee \neg E)$	#1	$(\neg B \vee E), (\neg D \vee C)$
#3	B=0	$(\neg B \vee E)$	#2	$(\neg A \vee B), (\neg D \vee C)$
#4	A=0	$(\neg A \vee B)$	#3	$(\neg D \vee C)$
#5		(A)	#4	violated clause

Explanation: *Feature D is dead, because: A is the root (#5), B is a mandatory child of A (#4), ¬B ∨ E is a constraint (#3), E and D are alternative children of C (#2).*

Table 3.3: Explaining the dead feature D.

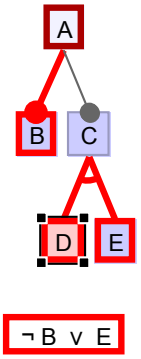


Figure 3.13: Result

Explaining False-Optional Features. False-optional features are modeled as part of alternative- and or-groups or simply as optional, but are always present if their parent is as well. The adaption of the BCP algorithm is intuitive: By setting the truth value of a false-optional feature to *false* and the direct parent feature to *true*, a contradiction will occur. The CNF must contain the constraint, which leads to a false-optional feature, since no violation would occur otherwise. An example for a false-optional feature is shown in Fig. 3.11c. Table 3.4 demonstrates the constraint propagation of the BCP algorithm. Setting the false-optional feature D to *false*, and its parent B to *true*, feature C must be *false* to satisfy the constraint $(\neg C \vee D)$. This is a contradiction to the clause $(\neg B \vee C)$ [KAT16, Ana16]. The final result is visible in Fig. 3.14.

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B) \wedge (\neg C \vee D)$

ID	Con.	Reason	AC	Stack
#1	D=0	premise		
#2	B=1	premise		$(\neg C \vee D), (\neg B \vee C), (\neg B \vee A)$
#3	C=0	$(\neg C \vee D)$	#1	$(\neg B \vee C), (\neg B \vee A)$
#4		$(\neg B \vee C)$	#3	violated clause

Explanation: *Feature D is false-optional, because: C is a mandatory child of B (#4) and ¬C ∨ D is a constraint (#3).*

Table 3.4: Explaining the false-optional feature D.

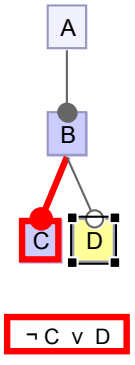


Figure 3.14: Result

Explaining Redundant Constraints. A CTC is only redundant, if the relationship among its features is already modeled in some other way in the feature model. We can use this relationship to explain the redundant constraint. First input is the generated CNF from the feature model without the redundant constraint. Of course, the CNF still includes the information contained in the CTC due to its redundant nature. However, choosing the premises is a bit more challenging for this defect. A contradiction only occurs if the truth values result in a non-satisfiable constraint. We can have multiple assignments leading to a non-satisfiable constraint. It is necessary to analyze all non-satisfiable combinations as individual explanations may give us an incomplete explanation. Each combination can result in a different explanation and only their union gives us a fully fledged solution. Duplicate explanation parts can be ignored as they provide no additional information in understanding the cause for a redundant constraint.

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B)$				
ID	Con.	Reason	AC	Stack
#1.1	B=0	premise		
#1.2	C=0	premise		$(\neg D \vee B), (\neg A \vee B)$
#1.3	D=0	$(\neg D \vee B)$	#1.1	$(\neg A \vee B)$
#1.4	A=0	$(\neg A \vee B)$	#1.1	
#1.5		(A)	#1.4	violated clause
#2.1	B=0	premise		
#2.2	C=1	premise		
#2.3		$(\neg C \vee B)$	#2.1	violated clause
#3.1	B=1	premise		
#3.2	C=0	premise		
#3.3		$(\neg B \vee C)$	#3.1	violated clause
Explanation: *Constraint $C \wedge B$ is redundant, because: A is the root (#1.5), B is a mandatory child of A (#1.4) and C is a mandatory child of B (#2.3, #3.3).*				

Table 3.5: Explaining the redundant constraint $C \wedge B$.

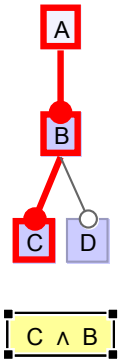


Figure 3.15: Result

For instance, we consider the feature model in Fig. 3.11d. The constraint $B \wedge C$ is redundant since B and C will always appear even without the constraint. The CNF of the feature model without that constraint is shown in Table 3.5. We can compute three different assignments that lead to an invalid formula. Table 3.5 presents the results computed by BCP. First, variables B and C are both bound to *false*. The unit-open clause $(\neg D \vee B)$ is removed from the stack and examined. Since B is

bound to *false* and the clause has to be satisfied, variable D is concluded to be *false*. The algorithm continues until a value is set resulting in a violated clause of the CNF, which is the case for the clause (A). In the second iteration, variable B is set to *false* and C is set to *true*. A violation in the CNF clause $(\neg C \vee B)$ appears because all terms are *false*. The third iteration sets variable B to *true* and C to *false*. A violation in clause $(\neg B \vee C)$ appears as all terms are *false*. Gathering the reasons for the three iterations results in the following set of clauses:

$$\{(A), (\neg A \vee B), (\neg C \vee B), (\neg B \vee C)\}$$

Although the clause $(\neg D \vee B)$ is existent in the reasons, it is skipped since the explanations are generated backwards based on the antecedents. The reason in iteration #1.5 refers to #1.4 and #1.4 directly relates to #1.1 as antecedent skipping #1.3 and the clause $(\neg D \vee B)$ for this explanation. Table 3.5 and Fig. 3.15 show the final explanation [KAT16, Ana16].

Explaining Implicit Constraints. Since an implicit constraint simultaneously is a redundant one, the actual BCP process is also similar comprising the same input parameters and premises leading to a non-satisfiable constraint. However, an explanation can involve features from multiple partial models, which is why the CNF of the sliced feature model is not sufficient to generate a full explanation. Although implicit constraints consist only of features from the partial model, their reason of existence always lies within the complete feature model due to the global CTCs. Again, consider the feature models in Fig. 3.6a with the implicit constraint $C \Rightarrow D$. Table 3.6 presents the BCP process explaining the implicit constraint $C \Rightarrow D$. First, we pass the complete feature model CNF and premises to BCP. In this case, we have only one truth value assignment leading to a violated clause with $D = false$ and $C = true$. We refrain from showing the complete CNF due to its length. BCP concludes G to be true and updates all respective variables. This results in a violated clause $\neg G \vee D$ [AKTS16]. A visualization for this defect is currently missing and explanations are purely given in textual form (cf. Sections 3.4.1 and 3.6).

ID	Conclusion	Reason	Antecedents	Stack
#1	D=0	premise		
#2	C=1	premise		$(\neg C \vee G)$, $(\neg G \vee D)$, $(\neg C \vee B)$
#3	G=1	$(\neg C \vee G)$	#2	$(\neg G \vee F)$, $(\neg G \vee \neg H)$, $(\neg G \vee D), (\neg C \vee B)$
#4		$(\neg G \vee D)$	#3	violated clause

Explanation: *Constraint $C \Rightarrow D$ is implicit, because: $G \Rightarrow D$ is a constraint (#4) and $C \Rightarrow G$ is a constraint (#3).*

Table 3.6: Explaining the implicit constraint $C \Rightarrow D$ [KATS17].

Characteristics of BCP. Overall, the BCP algorithm meets previously defined requirements. Since BCP works on the basis of predicate logic [FK93] and every feature model can be mapped to a propositional formula, BCP is *generic* and can explain the described defects. It has been invented several decades ago and was initially used to *efficiently* implement artificial intelligence [FK93]. The BCP algorithm only maintains reasons leading to inferences and ignores clauses, which do not contribute to a violation during the propagation. The additional tracing of CNF clauses to their feature model origin provides us with the information to create user-friendly explanations. A combination of both aspects leads to *informative* explanations for the developer on a textual or graphical level [KAT16].

3.3.3 Limitations

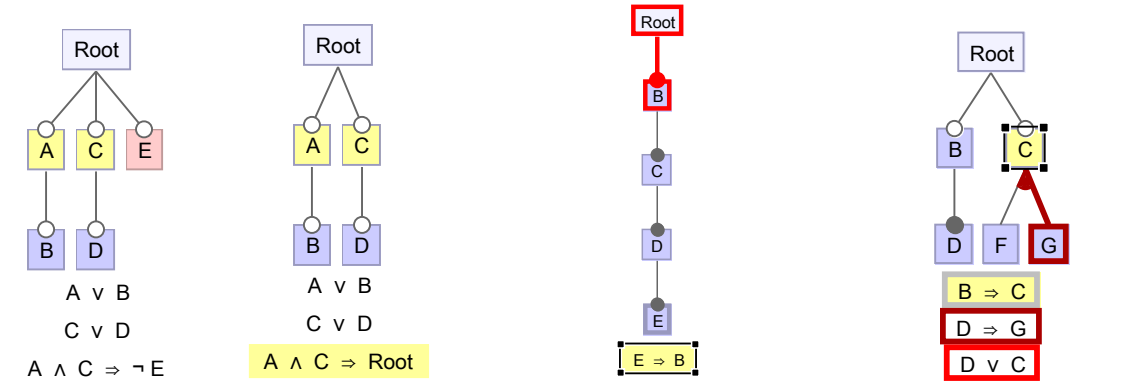
Although BCP is quite powerful and fulfills our requirements, we still encounter some limitations that we address in the following. The two main drawbacks are the *order-sensitivity* and *incompleteness* of BCP.

Order-sensitive. The generated explanation of BCP is influenced by the order of clauses in the CNF and the stack. For example, traversing the CNF from the left to the right as in all previously described examples may lead to a different explanation than processing the CNF from the right to the left. A similar observation can be made for the stack, since BCP originally infers always the latest unit-open clause on top of the stack. This limitation ultimately leads to the conclusion that BCP does not always find the explanation with a minimal length.

Incomplete. BCP can generate explanations for most of the typical scenarios in feature modeling. However, we encountered some rare cases in which BCP is not able to infer truth values. Fig. 3.16a and Fig. 3.16b depict two of such examples. BCP cannot generate an explanation for the dead feature E or the redundant constraint $A \wedge C \Rightarrow Root$. The problem lies within the unit-open clauses, since the propositional formulas of both examples do not contain any unit-open clauses. Hence, it is impossible for BCP to generate an explanation [FK93]. This behavior affects all possible defects and some more cases are given in the Appendix. Integrating alternative approaches, e.g., finding a minimal unsatisfiable core (cf. Chapter 3.5) may solve this problem. However, BCP remains a sophisticated first step in the generation of defects.

3.3.4 Improvements to the Generated Explanations

While the incompleteness of BCP remains a challenge, we can lift the problem of its order-sensitivity by introducing two improvements to the core algorithm. The



(a) Dead Feature (b) Redundant Constraint (c) Short Explanations (d) Emphasizing Parts

Figure 3.16: Limitations and Improvements [KAT16]

generation of a shorter explanation is possible without iterating through all clause combinations in the CNF or stack. We can take advantage of multiple generated explanations in order to emphasize core explanation parts.

Finding Shorter Explanations. In general, a short explanation is preferred by many developers, since the considered defect often is easier to understand, especially if we take large-scale feature models with several thousand features and constraint into account in which the explanation length may also rise. In the best case, we would like to find the minimal explanation for a defect. In order to guarantee this, BCP requires an examination of every possible order of clauses, which is not efficient in terms of computation times. Thus, we introduce a heuristic that improves the classical BCP algorithm and is based on the stack maintaining the unit-open clauses.

The basic BCP algorithm terminates after identifying a contradiction. In many cases the stack still contains unused unit-open clauses at this point, which can be used to generate additional explanations. Thus, we execute BCP again until the stack is empty. Consider Fig. 3.16c and the redundant constraint $E \Rightarrow B$. BCP generates the following two explanations with:

- *Constraint $E \Rightarrow B$ is redundant, because:*

1. Explanation	2. Explanation
• *E is a mandatory child of D,*	*A is the root*
• *D is a mandatory child of C,*	*B is a mandatory child of A*
• *C is a mandatory child of B.*	

Comparing the old algorithm (left-side) to the new improved one (right-side) favors the heuristic. Both explanations comprise only relevant information to comprehend the cause of the defect. However, it is not necessary to report the transitive chain which arises from the redundant constraint. The core nature of B is sufficient enough.

Emphasizing Core Explanation Parts. Given the previous improvement, BCP can already generate multiple explanations for one defect. We take advantage of this fact in order to emphasize the most relevant parts in an explanation. Every explanation is comprised of either parent-child relationships in the feature tree and/or CTCs. Some of these relationships and CTCs may appear in multiple explanations for a defect. Identical parts are more likely to be responsible for the case of a defect. Thus, editing these parts has a higher probability to repair a defect.

For instance, editing a CTC that is not present in all explanations cannot fully repair a defect, as at least one explanation containing another cause still remains for the defect. Fig. 3.16d shows a feature model including three CTCs resulting in a false-optional feature C. BCP generates the following three explanations for this defect using the heuristic [KAT16, Ana16]:

1. *G is an or-child of C, $D \Rightarrow G$ is a constraint and, $D \vee C$ is a constraint.*
2. *$D \vee C$ is a constraint, D is a mandatory child of B and, $B \Rightarrow C$ is a constraint.*
3. *$D \vee C$ is a constraint, $D \Rightarrow G$ is a constraint and, G is an or-child of C.*

Comparing the three explanations above leads to the conclusion that *$D \vee C$ is a constraint* appears in all explanations. Indeed, we can fix the defect by removing this constraint. Such information can be visually highlighted within an explanation in order to point out concrete parts that are more likely to cause the defect. Fig. 3.16d shows how the emphasis of crucial parts in explanations works using a different color-intensity, which ranges from black to red. Black parts are only present in one explanation, while red parts occur in all explanations [KAT16].

Coloring the Feature Diagram. The first realization of BCP and all of its concepts was purely based on textual explanations. Thus, a developer received an explanation without any visual feedback except for the colored emphasis of most relevant explanation parts. It is obvious that explanations containing more than 10 or 20 parts are difficult to comprehend even for domain experts. The origin of a specific clause, either feature tree or CTC, is saved during the creation of the CNF. Instead of just providing textual explanations with this information, we can directly color the feature tree, which was implemented in the second realization.

3.4 Evaluation

The evaluation is divided in two main parts. After a description of our prototypical implementation in FeatureIDE[1] [KTS+09], we focus first on the number and structure of implicit constraints and second on qualitative and quantitative aspects of our generated explanations. The considered case studies comprise the PPU, the simplified car feature model (cf. Fig 3.1) in detail as well as additional large-scale SPLs to show the applicability and scalability of BCP and our user-friendly explanations. In particular, we investigate the following research questions for implicit constraints:

- **RQ 3.1**: *How many implicit constraints exist in decomposed SPLs?* There is no significant amount of research available considering the existence of hidden dependencies. Hence, we want to prove that such dependencies do exist in this context and have to be addressed adequately.
- **RQ 3.2**: *How long does it take to derive implicit constraints?* The performance impact for the derivation of implicit constraints is necessary to investigate.
- **RQ 3.3**: *What is the structure of an implicit constraint?* Inspecting the implicit constraints and their explanations can reveal important information about the structure of the SPL.
- **RQ 3.4**: *How many partial models are involved?* Given the structure of such constraints, we can reason about the amount of local features from the observed submodel in comparison to the number of features from adjacent models gives us an even better insight into the SPL.

Considering the qualitative and quantitative aspects of explanations, we aim at answering the following research questions:

- **RQ 3.5**: *Do explanations contain the necessary parts to understand the defect?* An explanation is comprehensible, if changing or removing the generated parts can repair the defect. We also inspect the usability of the emphasized parts.
- **RQ 3.6**: *What is the average length of an explanation?* The length is a significant factor for the developer. Thus, we inspect the average length in feature models of different size and what the impact of the proposed heuristic, i.e., finding a shorter explanation, is for all explanations.

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

3.4.1 Implementation

We provide a prototypical implementation in the open-source framework FeatureIDE, which was released in version 3.1.0. The coloring of the feature tree followed in version 3.3.0.[2] The implementation of the adapted BCP uses the CNF generated from a feature model to infer reasons for a defect. An explanation is built by the combination of several reasons. However, such an explanation only consists of pure CNF clauses leading to the contradiction in the BCP algorithm (cf. Section 3.3.2). The information about CNF clauses is hardly beneficial for the developer, as their relation to the feature model is not obvious. We focus on the challenging task to provide the developer with useful feedback, which is why we trace the relations between the feature model and its CNF clauses.

Every feature contains structural information in a feature model. It occurs as part of the tree topology and can additionally appear in an arbitrary number of CTCs. Regarding the tree topology, a feature can take up different roles with *parent* or *child* and in addition it is defined as *mandatory*, *optional* or part of *alternative* and *or* groups. User-friendly explanations can be generated by reusing this structural information in combination with the concluded reasons of BCP. The mapping of a clause to the feature model comes with several difficulties [KAT16, Ana16]:

- Translating the feature model into a CNF results in a one-to-many relationship between a feature and the boolean variables representing this feature in the formula. A variable occurs multiple times in the CNF. Hence, every variable has to carry its structural information whether it is a parent, child, mandatory and so on in the tree topology or is contained inside a CTC.
- This annotation of a variable has to be as efficient as possible, since FeatureIDE operates with variables in multiple processes.

The annotation of every variable with the structural information takes place during the creation of the CNF of the feature model. FeatureIDE itself already provides means to retrieve the structural information of a feature. Furthermore, it includes the detection of defects with a SAT solver, which we reuse here (cf. Section 3.1.2).

After modeling a feature model in FeatureIDE, our approach can be executed by simply clicking on a defect. The explanation is instantly generated and the feature tree colored for this defect. In the first realization, we generated explanations for all defects at once and only in textual form. This approach had a significant performance impact especially in large-scale feature models [KAT16]. The second implementation has no noticeable performance drawback. A developer instantaneously sees the result. We implemented a lazy computation that generates only an explanation for the selected defect on-demand. The main tool support was developed in two student theses by Sofia Ananieva [Ana16] and Timo Günther.[3]

[2] https://github.com/FeatureIDE/FeatureIDE

[3] `https://www.youtube.com/watch?v=0n-CibotBnc`

The derivation, highlighting and explanation of implicit constraints works in a similar fashion. Developers have to right-click on a feature and select the context entry *Show Hidden Dependencies of Submodel*. A dialog opens containing the partial feature model with the selected feature as new root. Below the root, all features appear in the same way as in the complete feature model. Global CTCs relevant for the partial model are depicted below the feature hierarchy. Implicit constraints can be distinguished by a surrounding red border and are marked as redundant as well. The presented partial model is not editable at the moment. Reflecting changes in the partial model back to the complete model is a challenge for future work. The generated explanations for the implicit constraints also rely on the textual representation. The feature tree is not colored for this contribution, which is also left for future work.

3.4.2 Evaluating Implicit Constraints

The first research question tackles the actual existence of implicit constraints in feature models in general. First of all let us return to our running example, the PPU, for which our approach derives four implicit constraints with [KATS17, AKTS16]:

1. $Diagnosis \vee \neg Self\text{-}healing$ (Customer Feature Model)
2. $\neg Small \vee \neg Large$ (Customer Feature Model)
3. $Cylinder \vee ChangeOverArm$ (Electrical Feature Model)
4. $Diagnosis \vee \neg Self\text{-}healing$ (Software Feature Model)

Table. 3.7 presents explanations for the implicit constraints above retrieved using our extension of FeatureIDE. The first constraint is a hidden implication in the customer feature model because of two global CTCs. A similar dependency is detected for the software feature model shown in the fourth constraint. The explanations reveal that almost identical features are involved in both cases. Recalling Fig. 3.7, the explanations can be mapped to the last three CTCs resulting in two implicit dependencies. A hidden exclusion is expressed in the second constraint. We have two features in the customer model implying different features in an alternative relationship of the mechanical model. The third constraint gives the most complex explanation in our running example. Considering only the electrical feature model without implicit constraints, it is possible to avoid the selection of the *Cylinder* and *ChangeOverArm* at all. However, the derived implicit constraint forbids this configuration, since we have to select at least one of both features. We observe a transitive chain in the customer feature tree from the root *PPU* of the complete model to the two workpiece types *Small* and *Large*. Again, both imply different features even

from different feature models. However, due to bijections, an implicit dependency occurs in the electrical feature model.

The last column in Table 3.7 depicts all involved partial feature models for each implicit constraint. For the PPU, we mostly have two partial models responsible for a hidden dependency. However, in one case features from all partial models are present in the explanation. A complete explanation often involves features from multiple partial models, which is necessary to fully understand the hidden dependency and supports the communication between developers, since it is now obvious which features are responsible.

Next, we focus on performance measurements and evaluations concerning the number of implicit constraints in a large product line to fully answer **RQ 3.2** and **RQ 3.1**. As case example, we have a feature model from the automotive industry with 2,513 features and 2,833 global CTCs. The execution time has been measured using an Intel(R) Core(TM) i7-4800MQ CPU with 2.7 GHz and 16-GB RAM.

Table 3.8 presents the detailed results of our evaluation. The first column refers to the depth of a feature in the complete model that is selected as the root of the partial feature model. For instance, the automotive example has six child features below its root at depth one. The third column contains the number of features in these partial models, e.g., the largest model at position 5 contains 2,065 features in its subtree. This partial model also contains with eleven most of the implicit constraints at this depth. Overall, there are twelve hidden dependencies, which seems to be a rather small amount in comparison to the size of the complete feature model. Furthermore, we evaluate the child features of these six features as well resulting in 25 analyzed partial models. The number in brackets indicates to which parent feature (1-6) the 25 features belong. We can observe that the amount of implicit constraints significantly increased with 189. The computation time for partial models without any implicit constraints is 0.44 s on average, which is the time needed by the slicing algorithm to derive the partial model. Of course, our approach does not start any explanation attempts in these cases. Depending on the number of derived implicit constraints, the computation time rises with a linear factor. However, this is to be expected and the time with at most 170 s for 123 constraints is still acceptable. The average number of explanation parts (or length of the explanation) is 15.1 parts at depth one and 16.46 at depth two. Given several hundred features even in the partial models, we believe that the explanation is still comprehensible and definitely beneficial for the developer in understanding the dependencies. A more detailed reasoning on the explanation length is available in the next part of the evaluation. Thus, we conclude that implicit constraints are present and relevant for real-world feature models (cf. **RQ 3.1**) and the computation time is acceptable (cf. **RQ 3.2**) [AKTS16].

Table 3.7: Explaining implicit constraints of the PPU [AKTS16]. FeatureIDE does not allow two features with identical names in a single model making the suffixes S and E necessary. S represents a feature in the software model, while E stands for the electrical model.

Constraint	Explanation	Partial Models
Diagnosis $\vee \neg$ Self-healing	Self-healingS $\rightarrow$ Diagnosis is a constraint Self-healing $\rightarrow$ Self-healingS is a constraint	Customer, Software
$\neg$ Small $\vee \neg$ Large	Large $\rightarrow$ Cylinder is a constraint Cylinder and ChangeOverArm are alternative children of Lifting/Lowering Small $\rightarrow$ ChangeOverArm is a constraint	Customer, Mechanical
Cylinder $\vee$ ChangeOverArm	PPU is the root PPU_Customer is a mandatory child of PPU Work Pieces is a mandatory child of PPU_Customer Size is a mandatory child of Work Pieces Small and Large are or children of Size Small $\rightarrow$ ChangeOverArmE is a constraint Large $\rightarrow$ Cylinder is a constraint CylinderControl $\Leftrightarrow$ Cylinder is a constraint Cylinder $\Leftrightarrow$ CylinderE is a constraint	Customer, Mechanical, Electrical, Software
Diagnosis $\vee \neg$ Self-healingS	Diagnosis $\rightarrow$ DiagnosisS is a constraint Self-healingS $\rightarrow$ Diagnosis is a constraint	Customer, Software

Depth in FM	Partial Model (Parent)	# Features	# IC	Computation Time (s)
1	1	105	1	0.81
1	2	171	-	0.46
1	3	54	-	0.52
1	4	112	-	0.51
1	5	2065	11	542.60
1	6	5	-	0.43
2	7 (1)	8	-	0.50
2	8 (1)	72	-	0.47
2	9 (1)	4	-	0.48
2	10 (1)	3	2	1.05
2	11 (1)	17	-	0.45
2	12 (2)	167	-	0.48
2	13 (2)	3	-	0.45
2	14 (3)	21	2	2.08
2	15 (3)	18	-	0.46
2	16 (3)	3	-	0.46
2	17 (3)	3	-	0.44
2	18 (3)	5	-	0.44
2	19 (3)	3	-	0.41
2	20 (4)	3	-	0.46
2	21 (4)	88	-	0.45
2	22 (4)	16	-	0.43
2	23 (4)	4	-	0.45
2	24 (5)	684	123	170.83
2	25 (5)	16	-	0.44
2	26 (5)	948	39	75.7
2	27 (5)	231	20	21.24
2	28 (5)	185	3	2.51
2	29 (6)	2	-	0.45
2	30 (6)	1	-	0.44
2	31 (6)	1	-	0.46

Table 3.8: Implicit constraints for the automotive model [AKTS16]. *FM* = feature model, *IC* = implicit constraint.

RQ 3.3 questions the structure of the derived implicit constraints. We were able to identify three major groups of logical expressions:

1. Implication: The implicit constraint represents an implication, e.g., $A \Rightarrow B$ or $A \wedge B \Rightarrow C$.
2. Exclusion: The implicit constraint represents a mutual exclusion between features, e.g., $\neg(A \wedge B)$.
3. Negation: The implicit constraint represents a negated feature, e.g., $\neg A$.

Table 3.9 presents our classification of the derived 201 implicit constraints into the three major groups and an additional group for any other kind of logical expressions. We also give examples of the identified CNF patterns. As indicated by the overall percentage, almost all implicit constraints can be mapped to one of the major categories. Implication forms by far the largest group with about 75%. Finally, we took a closer look to the features occurring in the constraints (cf. Table 3.10). Naturally, an implicit constraint involves additional partial feature models meaning that some features in the constraint are part of another submodel. We observed that up to four individual submodels are involved in an implicit constraint. Less than 40% of all features in an explanation are local features meaning features of the partial model in which the implicit constraint occurred. An explanation consists of 24% up to 56.1% local features. On average, explanations comprise 37.35% local features leading us to the conclusion, that a hidden dependency is mostly caused by relations between features from other submodels, thus answering (**RQ 3.4**) [AKTS16].

Logic	CNF Pattern	Overall
Negation	$\neg A$	8.6 %
Implication	$\neg A \vee B$	14.6 %
	$\neg A \vee \neg B \vee C$	58.1 %
	$\neg A \vee B \vee C$	2.5 %
Exclusion	$\neg A \vee \neg B$	15.7 %
Other	$A \vee B \vee C$	0.5 %

Table 3.9: Classification of implicit constraints in logical expressions and representation as CNF patterns for the automotive model [AKTS16].

Consequently, implicit constraints do exist in a significant number and are necessary to understand partial feature models in isolation. Investigating the usefulness of our explanations is part of the next section.

3.4.3 Evaluating Explanations

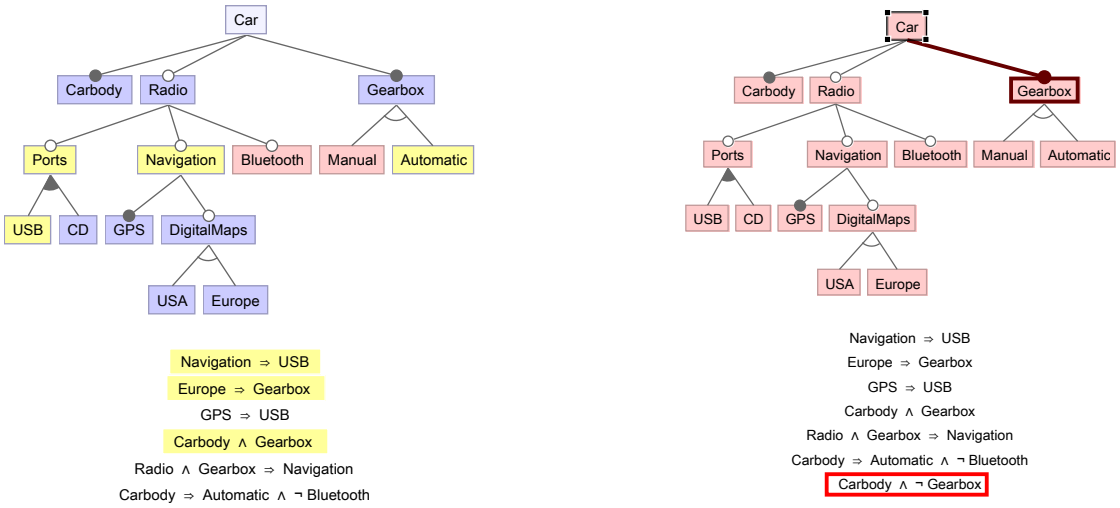
The second part of the evaluation was conducted with all available case studies from the PPU over the simple car feature model to the large automotive SPL with more than 2,000 features.

Depth	Partial Model	Involved Models	Local Features
1	5	3/6	40 %
2	14 (3)	1/25	25 %
2	24 (5)	2/25	29.7 %
2	26 (5)	1/25	24 %
2	27 (5)	1/25	56.1 %
2	28 (5)	4/25	51.3 %

Table 3.10: Analyzing explanations for implicit constraint with respect to further involved partial feature models and local features for the automotive model [Ana16].

Qualitative Evaluation. In particular, the comprehensibility is investigated using the simple car feature model. **RQ 3.5** is answered successfully, if we can understand and repair all detected defects with the generated explanations.

Void Model. As described in Chapter 3.1.1 and again depicted in Fig. 3.17a, the simple car feature model is not void. However, for the purpose of our evaluation, we temporarily add the CTC $Carbody \wedge \neg Gearbox$ to the feature model making it void. The explanation in Fig. 3.17b reveals that exactly the added constraint is identified as cause for the void model. Although multiple explanations are generated as indicated by the color gradient, explanation parts in a bright red color are present in all explanations and thus more likely to cause the defect. It is obvious that removing this constraint again would repair the defect.



(a) Originial feature model

(b) Void Model

Figure 3.17: Explaining a void model.

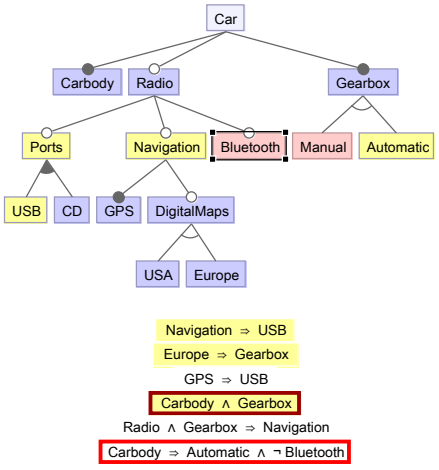
Dead Features. The *Car* feature model has two dead features, namely *Bluetooth* and *Manual*. Fig.3.18a and Fig.3.18b show the respective explanations. Due to the core feature *Carbody*, we must select an *Automatic* transmission. *Automatic* itself is part of an alternative group and therefore excludes all other features in this group, here only *Manual*. *Bluetooth* is dead, since its negation is implied by a core feature.

False-Optional Features. Fig. 3.18c-3.18f show the explanations generated for the four false-optional features *USB*, *Ports*, *Navigation* and *Automatic*. The explanation for the feature *Automatic* is quite similar to the one of the dead feature *Manual* due to the constraint $Carbody \Rightarrow Automatic \wedge \neg Bluetooth$. *Navigation* is always selected if we have a *Radio* and implies itself *USB* which is a child of *Ports* making all three false-optional. The CTCs and feature tree dependencies responsible for this behavior are fully revealed in the generated explanations making the defects easy to comprehend.

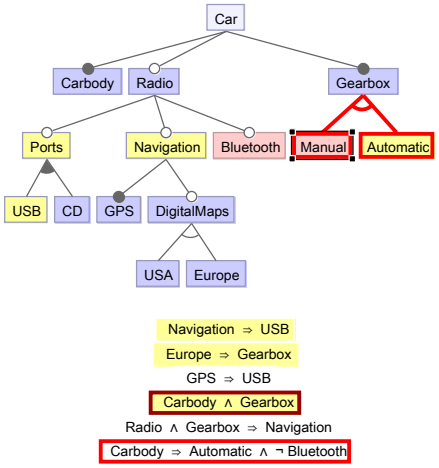
Redundant Constraints. Overall, there are three CTCs marked as redundant in the feature model. $Carbody \wedge Gearbox$ is redundant, because both features are already core features (cf. Fig. 3.19c). Among all explanations, we always present the improved one concerning length and coloring. The CTC $Europe \Rightarrow Gearbox$ is redundant as implying a core feature is meaningless (cf. Fig. 3.19b). In case of the last example $Navigation \Rightarrow USB$, the redundancy is caused by the constraint $GPS \Rightarrow USB$. By selecting the *Navigation*, it is already implied that the car has a *USB* port due to the mandatory feature *GPS*. Hence, it is not necessary to imply *USB* again with the *Navigation* feature.

To conclude the comprehensibility, we can observe that the explanations contain the required information to fix the detected defects. For the *Car* feature model, we have successfully shown the applicability of the presented algorithm as well as the satisfaction of **RQ 3.5**. We are aware that this example does not cover a full empirical evaluation of our approach with external developers. However, we postpone a detailed user study to further prove its benefits to future work [KAT16].

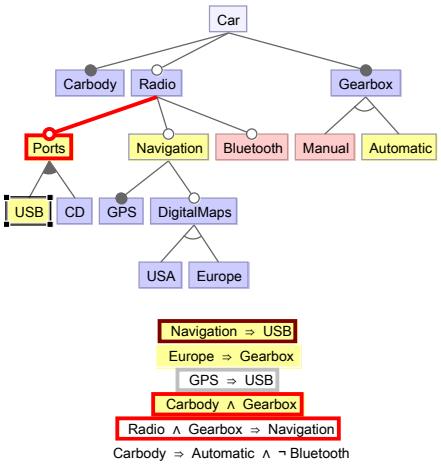
Quantitative Evaluation. As the next step, we focus on evaluations concerning the length of the short explanations to answer **RQ 3.6**. Table 3.11 presents the evaluated feature models along with information about their number of features, constraints and defects. The first feature model is our running example. Furthermore, additional generated feature models consisting of 200, 500, 1,000, and 2,000 features and constraints, respectively, were taken into account for the evaluation. These models have already been used in prior evaluations [KTS+09]. A feature model from the automotive industry represents the biggest feature model with 2,513 features and 2,833 constraints and it is our second real-world example. Void feature models were



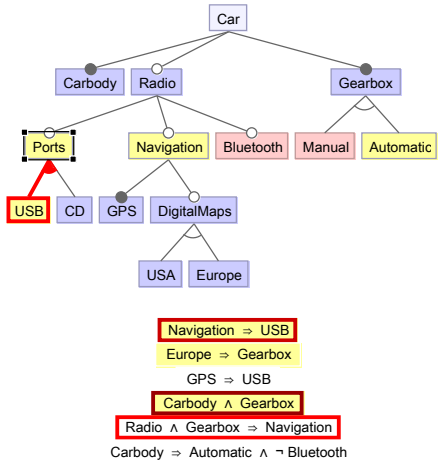
(a) Bluetooth is dead.



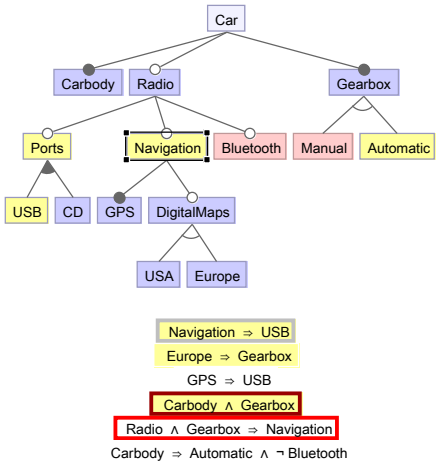
(b) Manual is dead.



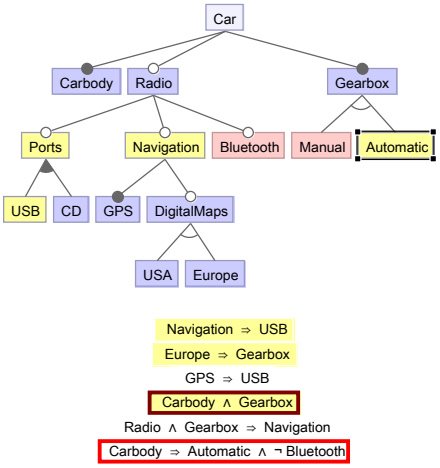
(c) USB is false-optional.



(d) Ports is false-optional.

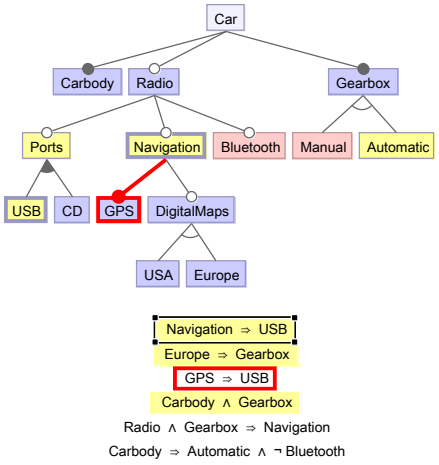


(e) Navigation is false-optional.

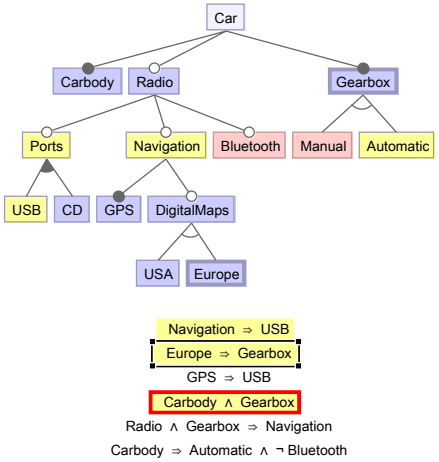


(f) Automatic is false-optional.

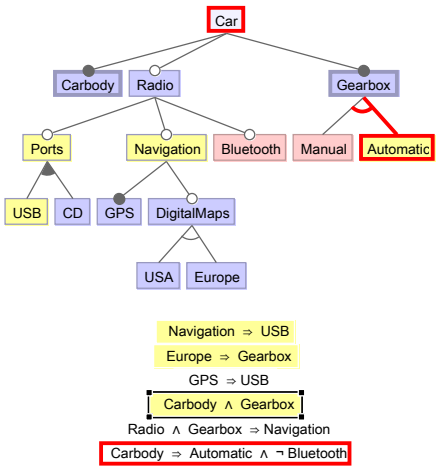
Figure 3.18: Explaining defects in the car feature model.



(a) Redundant Constraint 1.



(b) Redundant Constraint 2.

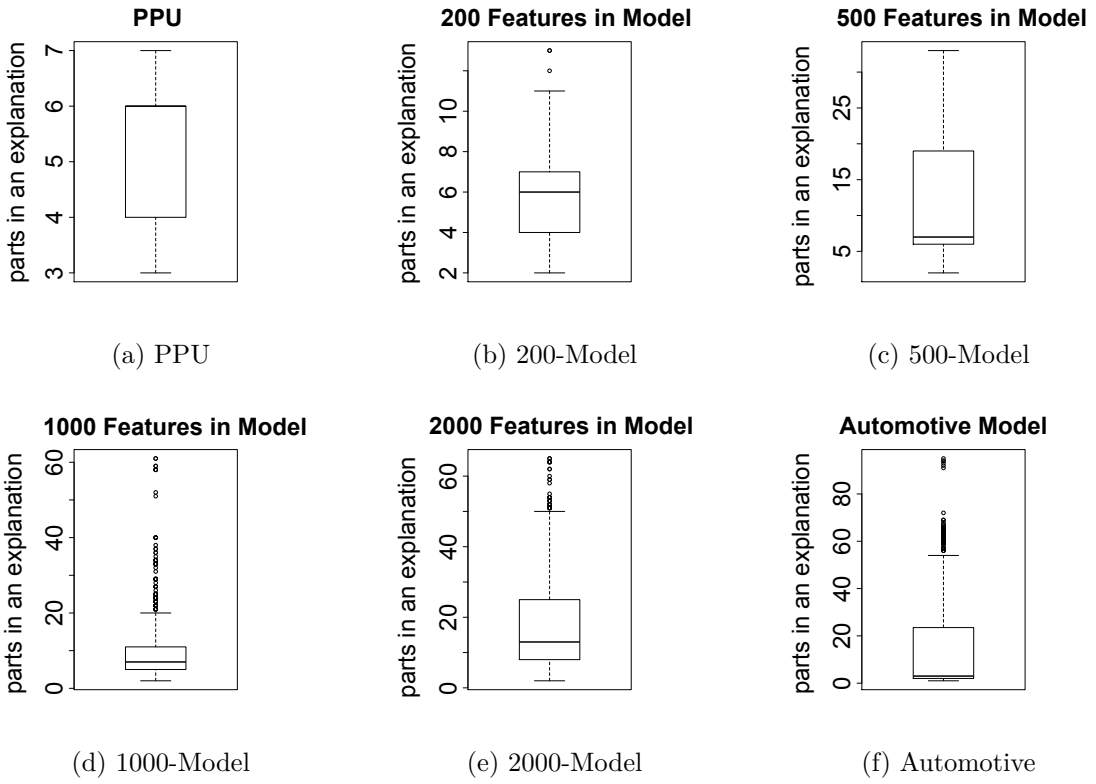


(c) Redundant Constraint 3.

Figure 3.19: Explaining defects in the car feature model.

Model	# Features	# CTCs	# RC	# D	# FO
PPU	52	15	5	0	0
200-Model	200	20	8	106	13
500-Model	500	50	14	262	56
1000-Model	1,000	100	44	628	138
2000-Model	2,000	200	87	1,236	254
Automotive	2,513	2,833	563	192	12

Table 3.11: Overview of evaluated feature models [KAT16, Ana16].



(a) PPU (b) 200-Model (c) 500-Model

(d) 1000-Model (e) 2000-Model (f) Automotive

Figure 3.20: Improved explanation lengths [KAT16, Ana16].

prohibited in the automatic generation process. The evaluation includes all detected defects in the feature models. The computation time is not considered here, since the performance impact is not perceivable by developers [KAT16]. We measure the explanation length in its number of parts. Fig. 3.20 illustrates the explanation length for all improved explanations found in the feature models. Given the rather small product line of the PPU, we observe that each explanation has at most seven parts (cf. Fig.3.20a). Although the next model is about four times larger, we only see a slight increase in the explanation length in Fig. 3.20b.
The explanation length continues to increase for the larger models as well. However, even for the automotive feature model 50% of the explanations consist of 4 to 25 parts [KAT16, Ana16].

While reasoning on the length of improved explanations, we can determine the frequency of a first explanation already being the shortest possible based on our algorithm. In Table 3.12, the number of explanations for redundant constraints per model is illustrated as well as how often an improved explanation has been found in later processing steps. Additionally, a relative shortening concerning the

Model	# Explanations	# Improved Explanations	Shortening (%)
PPU	6	1	44.4
200-Model	8	2	50
500-Model	14	2	25.1
1000-Model	44	11	39.7
2000-Model	87	21	48.4
Automotive	563	56	29.8

Table 3.12: Finding improved explanations [KAT16].

explanation length is presented. Given these results, we were able to make the following observations:

- The average explanation length grows only slightly compared to the number of features and constraints.
- In most cases, the adapted BCP finds a short explanation in the first run.
- Shorter explanations are between 25-50% smaller.

Considering the length of shorter explanations (cf. **RQ 3.6**), we conclude that even for large feature models a significant number of explanations stay below 20 parts, which is still comprehensible for developers, especially with regard to the developed visualization. Finally, we can define the size of a model as the number of features plus the number of CTCs. The maximum length of an explanation would be identical to the size of a model. A comparison of the model size to the average explanation length reveals that for the PPU about 10% of the model is involved in a defect, while this percentage steadily decreases in larger SPLs. For the automotive model less than 1% is present in the explanations on average [KAT16, Ana16].

3.4.4 Discussion

Next, we summarize the collected data and our observations in a condensed form regarding the defined research questions.

RQ 3.1: *How many implicit constraints exist in decomposed SPLs?*

We were able to prove the existence of hidden dependencies in feature models. Even relatively small SPLs such as the PPU contain implicit constraints. We identified four implicit constraints for the PPU. In realistic large-scale SPLs the appearance depends on the considered depth and of course heavily on the structure. If a large portion of features is clustered in a single partial model (cf. Table 3.8, Depth 1), we are not likely to have many implicit constraints. On the opposite, if we have a more even distribution of features, it is more likely to encounter implicit constraints (cf. Table 3.8, Depth 2).

RQ 3.2: *How long does it take to derive implicit constraints?*

The feature model slicing algorithm is very efficient with an average runtime of 0.44 s. It is not surprising that the computation time increases with the number of implicit constraints, but is still acceptable. Even for the automotive example the complete process takes nine minutes at most, thus a developer has enough time to grab a cup of coffee if a large partial model is under consideration.

RQ 3.3: *What is the structure of an implicit constraint?*

Implicit constraints occur most often as implications and exclusions with almost 91%. Thus, we have most likely similar cases to our introductory examples in Fig. 3.6.

RQ 3.4: *How many partial models are involved?*

Less than 40% of all features in an explanation are local features leading us to the conclusion that the interweaving of partial models plays an important role during SPL development. In case of the PPU, we could observe that in one case all partial models were involved (cf. Table 3.7).

RQ 3.5: *Do explanations contain the necessary parts to understand the defect?*

Except for the special cases in which no unit-open clauses exist, BCP was always able to generate an explanation. The explanations are understandable and support the repair process of the defects in case of the PPU and the simple car feature model. The presented results look very promising for larger SPLs. However, a more detailed investigation for large-scale SPLs is left for future work.

RQ 3.6: *What is the average length of an explanation?*

The explanation length is critical for the comprehensibility of a defect. First of all it stays constant at different depths in case we explain an implicit constraint. The best result definitely is the relation of the feature model size to the explanation

length, since the length increases only slightly compared to the model size. Considering the automotive model, we have to look at only 0.34% of the complete model given our explanations on average to comprehend a defect. However, there is always a possibility to generate exceptionally long explanations, e.g., at most 95 parts for the automotive model. Such explanations remain difficult to understand for developers. Our improved BCP algorithm generates multiple explanations for one defect. In most cases, BCP computes the shortest explanation immediately. Hence, the first explanation is already the shortest one. We also validated the scalability to large-scale SPLs.

3.4.5 Threats to Validity

Finally, we discuss possible threats concerning our case studies and methodology according to Wohlin et al. [WRH+12]. *Conclusion validity* ensures the rationality of our conclusions on a statistical level given the available data and its relationship. While conclusion validity focuses on whether there exists such a relationship, *internal validity* evaluates a causal relationship between both implementation and results. No unknown or not measured factor must be involved in the final result. *Construct validity* assesses whether we designed the evaluation correctly in order to retrieve the desired data. Finally, *external validity* is related to generalizability of the outcome, e.g., derived conclusions hold for other feature models.

Conclusion Validity. Due to the lack of realistic industrial-size feature models that are still scalable for automated analyses and available to the general public our results are based on a single model from this domain. A full validation of the conclusions made for implicit constraints, e.g., logic structure, occurrences and involved models, is only possible with further large-scale feature models. The threat also remains for the length of an explanation.

Construct Validity. The construct validity also suffers from the threat of just one large-scale feature model. Nevertheless, we carefully analyzed the model at different depths leading to a reasonable variety and amount of data.

Internal Validity. An explanation completely relies on the order of clauses in the CNF. Different CNF orderings may lead to different explanations. To counter this threat, we proposed a heuristic that also generates multiple explanations and may produce an even shorter one. A combinatorial exploration of all combinations is not reasonable due to its performance impact. However, a minimal explanation cannot be guaranteed at this point and calls for other approaches (cf. Section 3.5). In addition, artificially generated data, such as some of our feature models, always has a

possible distorting effect on the results. This is especially the case for the explanation length. However, these models have proven themselves in other evaluations such as the feature model slicing [KST+16] and are available from S.P.L.O.T.[4] We observed identical results to the automotive model as well.

External Validity. Again, the generalization suffers from the use of generated models and just a single realistic large-scale feature model. We argue that an adequate overview of the generated explanation length is present by covering a large number of different defects.

3.5 Related Work

In the following, we examine the existing work relevant for this contribution. First, we begin with the topic of detecting defects, which is a prerequisite for our proposed approach. Next, we focus on the actual explanation of defects. Last and major part of this related work section is dedicated to the decomposition of feature models and hidden dependencies.

Automated Analyses and Defect Detection. There has been a considerable amount of work on feature modeling and the automated analysis of feature models. In particular, many tools are available such as **FeatureIDE, TVL, FAMILIAR, SPLConqueror, Clafer,** and **pure::variants** [CBH11, MBC+13, SRK+12, BDA+14]. In addition, many approaches including the previously mentioned ones already provide support for the detection of defects [BSRC10, BFGR13, EPAH09]. An overview of analysis tools is given by **Meinicke et al.** [MTS+14]. The field of automated analyses spans to 30 different analysis operations that range from defect detection and explanation to product validation and refactoring [BSRC10]. **Von der Maßen and Lichter** [vdML04] defined various feature model defects from which we benefited in our work. **Hemakular** [Hem08] proposes an approach to detect defects in feature models with the SPIN model checker. The approach combines model checking with BCP in order to propagate consequences of a feature selection and report violations. In general, model checking is a powerful analysis method, since it can fully verify the considered model with respect to a specification. In this case, model checking identifies user selections leading to an invalid product. An invalid product results from a violation which is detected by BCP during constraint propagation, e.g., a dead feature. The approach is limited due to the difficulty of model checking large feature models. **Trinidad et al.** [TBD+08] also contributed a significant amount of research to the detection of defects formulating

[4] `http://www.splot-research.org/`

a constraint satisfaction problem as foundation. Since the concepts of **Trinidad et al.** involve explanations, we postpone a detailed discussion of their work to the related work on explanations. Our proposed approach takes advantage of an existing automated analysis for the detection of defects in the FeatureIDE framework. FeatureIDE is well-known and widely used in the SPL community even for industrial SPLs [LEGP15, SKT+16]. Thus, we argue that FeatureIDE is a sophisticated choice to form the foundation of our contribution.

Explaining Defects. The first attempt to connect propositional formulas to product lines was performed by **Mannion** [Man02]. **Batory** [Bat05] built upon this work and introduced LTMS and BCP to support developers in the configuration of a variant based on a feature model. The implementation is available in GUIDSL and provides feedback in terms of why a certain feature cannot be selected. In contrast to our work, GUIDSL can only explain dead features and focuses on the configuration process, while we already support the developer during development of the feature model (cf. Section 3.1.2, **Approach 1**). We improved Batory's work in different ways: First, by explaining more defects and also expressing explanations in a user-friendly way. Second, our explanations are closer to the feature model by referring to the structure and, hence, increasing the transparency of explanations. Third, we emphasize important explanation parts and show the scalability [KAT16].

A renowned algorithm concerning the explanation of a CSP is called *QuickXplain* [Jun04]. It uses of a divide-and-conquer strategy to compute a minimal set of faulty constraints in a CSP. **Lesta et al.** [LSW15] adapt the *QuickXplain* algorithm for the explanation of defects in attributed feature models. An attribute to a feature is similar to a variable in programming languages or the respective attribute in a UML class diagram. They are often based on integer values [LSW15, Jun04]. In order to keep the implementation efficient, the authors do not search for a minimal explanation, which is similar to our approach. We focus on explanations for classical feature models and considered no attributes. **Lesta et al.** explain all defects except for redundant constraints. Since the approach heavily depends on a specific constraint solver, it is difficult to apply other explanation strategies whereas our approach is independent and BCP can be replaced easily. Finally, the source code is not open-source [LSW15].

Returning to the work of **Trinidad et al.**, we can say that it is closely related to the contributions provided in this thesis. The authors use the *Theory of Diagnosis* invented by **Reiter** [Rei87] as foundation and retrieve a minimal set of constraints relevant for the explanation of a defect (cf. Chapter 3.1.2, **Approach 2**) [TBD+08]. The implementation is available in the FAMA tool [Tri12, BSTRC07]. Again, we focus on the modeling level and support the developer at an early development stage to repair a defect. As many other approaches FAMA supports the configuration

process by presenting a set of necessary feature selections and deselections to fix an invalid product configuration. Again, **Trinidad et al.** are able to explain all defects except for redundant constraints. In addition, the explanations are expressed in an abbreviated way which is at the expense of comprehensibility compared to our colored feature tree (cf. Section 3.1.2, **Approach 2**) [TBD+08].

Felfernig et al. [FBGR13] also demonstrate an approach based on the *Theory of Diagnosis* using the *FastDiag* algorithm. Similar to the previous QuickXplain algorithm, the FastDiag algorithm implements a divide-and-conquer strategy. The input consists of a set of constraints and it divides the constraints into subsets that ultimately contain the smallest subset of inconsistent constraints. The final explanation is presented in form of the responsible constraint set, which has to be deleted or adapted in order to repair the defect [FS10]. FastDiag can guarantee a minimal explanation and it is not restricted to a specific solver. Similar to our work, all defect types can be explained. Nevertheless, the evaluation is solely conducted with a feature model offered by the S.P.L.O.T. repository that contains only 172 features. In order to achieve the desired defects, CTCs have been randomly inserted to produce the inconsistencies. No information about the number or type of the defects is provided and the evaluation lacks a large-scale feature model to show the scalability. We argue that a combination of both approaches would be beneficial, since a minimal explanation paired with the proposed visualization has the most advantages for developers.

The ontological rule-based approach by **Ricón et al.** [RGMS14] poses the third explanation concept shown in Chapter 3.1.2. It generates explanations in natural language similar to our first realization. However, it can only explain dead and false-optional features. Similar to GUIDSL, **Kramer et al.** [KSRB13] present an approach to generate explanations for configuration purposes. In particular, they add explanation fragments as attributes to features and relationships while the concatenation of such fragments forms a complete explanation for a configuration. Although explanations are also expressed in a user-friendly way, the approach is only viable for small or medium sized feature models. In contrast, we give explanations during the modeling phase and strive to find short explanations. **Osman et al.** [EPAH08] extend feature models by so called variation points, before an explanation is possible. It was our goal to refrain from such extensions or to put additional workload on the developer.

Finding a guaranteed minimal explanation is also proposed by a few other approaches. **Liffiton et al.** [LS08] aim to find a minimal unsatisfiable subset (MUS) of clauses, hence the *core* of an unsatisfiable boolean formula. Mapping a MUS to defects in feature models results in a minimal explanation consisting of clauses in propositional logic. It is also possible to compute a minimal correction subset meaning the minimal set of clauses that must be removed in order to get a satisfiable

formula. **Liffiton et al.** developed a two-step approach called *CAMUS* (Compute All Minimal Unsatisfiable Subsets). As first step, CAMUS finds all minimal correction subsets and in a second phase it derives all MUSs. The authors state that the used algorithms can easily be adapted to work with different solvers making them independent. The evaluation of CAMUS proves its scalability and also reveals that it performs significantly better than other existing algorithms for the two phases. The main drawback of CAMUS is the resource cost of the first step making it not advisable if only one minimal explanation is required, i.e., a developer is only interested in one defect.

A state-of-the-art approach to determine a minimal explanation based on clauses is proposed by **Guthmann et al.** [GST16]. The concept is implemented in a tool named *HSMTMUC* and sits on top of a satisfiability modulo theories (SMT) solver. Similar to the previous approach it determines the minimal unsatisfiable core in a boolean formula [NRS13]. The core represents an explanation consisting of pure clauses if *HSMTMUC* is applied to the domain of explaining defects in a feature model. SMT solvers conclude the satisfiability for formulas in first-order logic and are capable of finding an unsatisfiable core [BSST09]. **Guthmann et al.** evaluated two SMT solvers with Z3 [DMB08] and MATHSAT [CGS07] to determine the unsatisfiable core and afterwards process it further in *HSMTMUC*. The evaluation shows that the best runtime and the smallest average core size is achieved by using MATHSAT and invoking HSMTMUC afterwards. As future work, one can exchange BCP with an algorithm that calculates the minimal unsatisfiable core and therefore always gives us the smallest possible explanation. In combination with the existing visualization, we would achieve an extraordinary solution for this challenge.

Finally, the analysis of a product line to determine defects is not limited to feature models only. Actual source code can also be analyzed, e.g., to find dead code blocks or compiler errors [TAK+14]. **Tartler et al.** propose techniques to detect anomalies in the source code of the Linux kernel [TLD+11, TLSSP11, TSSPL09]. An application of our work to explain dead code or compiler errors could help developers in fixing those problems, especially as the actual cause could also be in the feature model [KAT16].

Decomposition and Hidden Dependencies. Several concepts to reduce complexity for the developers in feature models exist in the literature. We decided to use interrelated feature models, since they were already available for our running example and supported by the FeatureIDE extension for VELVET [STSS13]. Another concept relies on the introduction of different views onto a feature model. Each view contains only the relevant information for itself. Instead of actually removing the features as in slicing, views hide undesired features. **Clark et al.** [CP10] provide a first theory for feature model views. Checking the compatibility of different

views as well as reconciliation of compatible views is possible. Views are often connected to the configuration process and not the actual development or maintenance of a feature model [SLW12, HHS+11]. Similar to the previous approach **Schroeter et al.** [SLW12] develop and evaluate a formalism for defining multiple views and checking the inconsistency. These aspects can be extended to the concept of multi software product lines combining several product lines and expressing their dependencies [ZKDT11]. The PPU can also be considered as a multi software product line and VELVET supports their development. VELVET also adapts the concept of merging the separate feature models together below an artificial root and lets stakeholders express dependencies between the individual models [STSS13]. **Lettner et al.** [LEGP15] added functionality in FeatureIDE to support different modeling spaces, modeling at different abstraction levels and dependencies between the spaces using inter-space dependencies. Inter-space refers to dependencies between the solution, problem, and configuration space based on the feature model. Revealing and understanding such dependencies between features from different spaces turned out to be extremely challenging. Although, we do not operate in different spaces, this challenge can be exactly mapped to dependencies in partial feature models.

An approach that detects hidden dependencies was presented by **Mendonça et al.** [MCMdO08] and works at the configuration level. The authors state that the configuration process is complicated due to multiple involved parties. Consequently, they propose a possible solution with collaborative product configuration to coordinate this process. The validation of products is supported by an efficient dependency analysis algorithm detecting interdependent relations, i.e., the selection of one feature must automatically select another feature. **Mendonça et al.** [MCMdO08] perform an efficient dependency analysis on graphs representing the feature models for the detection process. Contrary to our approach, interdependent relations are detected on configuration level, whereas we operate on modeling level. Additionally, no explanations of the detected dependencies are given.

Jonata and Botterweck [JB08] focus on describing an SPL from different perspectives, i.e., a feature model and an architectural model. The latter is a component model containing implementation details of the SPL. During the feature selection process, we may encounter inconsistencies between both models, e.g., a feature model allows a configuration which is not possible due to technical restrictions of the architecture. To solve this problem, the authors propose an integration of both models. For this purpose, the models are transformed into propositional logic and combined. An application of existential quantification enables the computation of a feature model containing such hidden dependencies. The original feature model is then extended with the derived dependencies. Thus, it can support the customer or user during the configuration process [JB08]. **Jonata and Botterweck** use existential quantification to produce a feature model with implicit constraints,

whereas the slicing algorithm by **Krieter et al.** [KST+16], which is implemented in FeatureIDE and reused in our work, takes advantage of logical resolution. Although logical resolution is also based on existential quantification, it is an improvement due to the preservation of the formula in CNF.

Ghanam et al. [GM10] also detect hidden dependencies between features. Features are typically linked to code artifacts in SPL development. **Ghanam et al.** use executable acceptance tests in their approach. These tests are specifications of defined requirements and can automatically test the behavior of a system. The authors extend features containing implementation artifacts with several executable acceptance tests. Thus, the selection of a feature forces the execution of all tests. Furthermore, the tests also inherit all dependencies of the linked features, i.e., tests for features in alternative groups are alternative as well. However, the hidden dependencies only comprise exclude and require relationships in this approach. After the detection, developers have to inspect the respective implementation artifacts as well as the feature model in order to identify the cause of a failed test. Contrary to our work, the approach cannot detect arbitrary implicit constraints [GM10]. Overall, no approach actually considers the explanation of hidden dependencies after the detection process. Thus, we argue that our proposed approach provides a novel contribution to this research field.

The removal of features from a feature model while preserving all existing dependencies is called feature model slicing. **Thüm et al.** [TKES11] introduced this concept during the reasoning on edits between two feature models. They were especially interested in the removal of abstract features [TBK09]. According to **Thüm et al.** editing a feature model produces a new model which can be classified into either a specialization (products are removed from the SPL), a generalization (new products are added to the SPL) or a refactoring (the set of products is maintained). In order to classify an edit, it is necessary to remove abstract features from both models. The authors delete every appearance of an abstract feature in the CNFs until it contains only concrete features. FeatureIDE provides tool support for the classification of edits [TBK09, TKES11]. Given this first idea, **Acher et al.** [ACLF11a, ACLF11b] implemented a feature model slicing technique with focus on the decomposition of feature models into multiple interrelated ones. The slicing algorithm is part of their FAMILIAR tool, which provides a domain specific language for feature model manipulation [MBC+13]. A rivaling slicing algorithm is developed by **Krieter et al.** [KST+16] several years later and integrated in FeatureIDE. We benefit from this implementation, which uses logical resolution at its core, and reuse it to derive implicit constraints in the partial feature models. In addition, the algorithm was also successfully used to derive so-called feature model interfaces [SKT+16]. A feature model interface contains an arbitrary subset of features and hides the remaining information similar to the previously described view.

Schröter et al. [SKT+16] provide means to enable the composition of multiple interfaces while keeping the computation cost for automated analyses operations low, since they can reuse results from individual interfaces.

3.6 Chapter Summary and Future Work

We have presented a generic and efficient algorithm for explaining defects in feature models based on propositional logic. It is able to explain most types of defects, which can be encoded in a CNF and a set of initial truth value assumptions. An additional tracing of literals to structural information of the feature model provides us with the means to express explanations in a user-friendly manner. In addition, we have presented an approach for deriving and explaining implicit constraints in interrelated feature models. Implicit constraints are indeed existent in real-world SPLs and cannot be neglected for development and maintenance purposes. The scalability is shown by analyzing several large-scale feature models including an industrial one. In all cases, the explanation length stays acceptable at no perceivable cost for the computation time. We implemented our approach as part of the open-source framework FeatureIDE [KAT16, AKTS16].

Regarding future work, the approach can be extended in several directions. The most beneficial improvement is the computation of minimal explanations for all defects. In addition, irrelevant parts of the model with respect to the explanation could be concealed in large feature models improving the comprehensibility of our explanations even further. A user study with industrial developers would investigate the actual saving of time using our explanations during development and maintenance. To reduce remaining threats to our evaluation, an extension in the number of analyzed large-scale feature models is necessary. Finally, an important task is to enable edit operations in the partial feature models and reflect them back to the complete one in order to fully support maintenance. A visualization similar to the other defects would be advantageous for implicit constraint as well [KAT16, AKTS16].

4 Multi-Perspective Modeling in the Solution Space

This chapter shares material with work published in [KLL+14], [KPST14] and [KS16].

Contribution

We devise a design-level modeling approach consisting of three modeling perspectives. Each modeling perspective provides a different viewpoint onto the system following the principle of separation of concerns. The models are equipped with variability modeling capabilities in order to support developers in creating reusable implementation artifacts. The approach includes textual and graphical model editors as well as a consistency checking concept. Its applicability is shown with our running example of the PPU.

Modern software systems are extremely long-lived. This is especially the case in the automation domain in which systems are operated for several decades. Due to limited durability of physical system components or technology changes during the system's lifetime, variability is introduced in the hardware parts. Software counterparts are successively affected as well [LBKVH12]. Besides, there are several other reasons for variability, such as differing production requirements, process improvements or product variations. Typically, there are different variants of the same automation system available [BBO+12], e.g., to satisfy varying customer needs. All of these aspects lead to a high diversity in modern automation systems creating an increased complexity for system management and maintenance [Bro95]. Ultimately, the proper functioning of the system has to be ensured for all variants [VHFF+15, KPST14].

In this chapter, we propose a multi-perspective modeling approach in order to model a system on different levels of abstraction. The approach is comprised of well-known models taken from the Unified Modeling Language (UML) [MG15]. In addition, each modeling perspective is equipped with delta modeling providing developers with the capabilities to create reusable artifacts and model all possible variants of the SPL in a similar fashion. We extended the concept to also identify inconsistencies across the variants that developers may have introduced by the usage of delta modeling.

Again, we first lay the necessary foundation for this contribution. Next, we in-

troduce the individual modeling perspectives as well as their connection. This part also includes the application of delta modeling to each modeling perspective and a concept for consistency checking. This is followed by implementation-specific details and an application of the presented ideas to our running example of the PPU. Afterwards, we compare our concept with already existing ideas in the literature. The chapter is concluded by a summary and discussion of points left for future work.

4.1 Preliminaries

The foundation consists of two topics. First, we point out the benefits of model-driven development compared to classical software development. Second, we introduce the considered UML diagram types used for our approach.

4.1.1 Model-Driven Development

> "Model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler." (Bran Selic, IBM Rational Software, 2003)

In Model-Driven Development (MDD), the primary artifacts of interest are models. A model is an abstraction of the reality, represents a real world object or situation [Sta73]. MDD continuously gets more attention, especially in the industrial domain [MCF03, Sel03, Vya13, TPK07], and several sophisticated tools have been developed in this context showing its benefits. A famous example is Matlab[1] with the Simulink extension which enables block-based development on a graphical level. Simulink itself contains packages to provide code generation and simulation capabilities based on these models. Overall, MDD improves the productivity in large-scale software systems compared to classical software development after outliving a first adjustment phase. In particular, the expected benefits are as follows [Amb08, AK03, FR07, Sch06, KLL+14]:

- A higher level of abstraction results in an easier and more understandable way to describe the system during all periods of development, since we can hide complex details.
- Lower redundancy increases productivity, shortens the development time and therefore reduces the costs for the development.
- Separation of concerns allows developers to focus on their fixed viewpoint/perspective and reduces the complexity for each person.

[1] `https://de.mathworks.com/products/matlab.html`

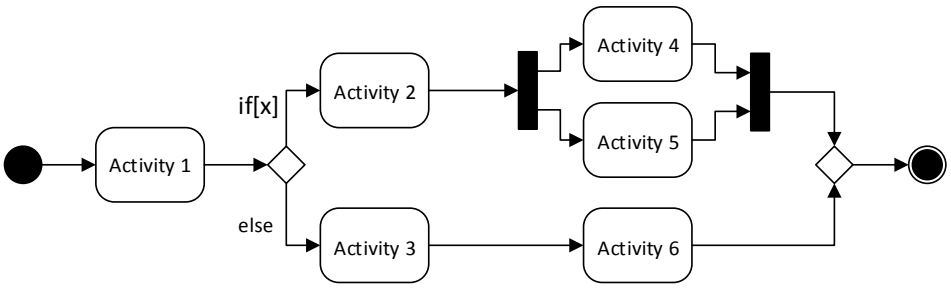
- Testing and simulation can be performed on a more abstract model level, also including automated analyses of models to prove the soundness and/or other important properties of complete models or some model parts.
- Automated code generation produces highly optimized and efficient source code in terms of required resources.

We want to leverage these benefits and also pursue an approach that primarily focuses on models as artifacts of interests and fully supports MDD with respect to SPLs.

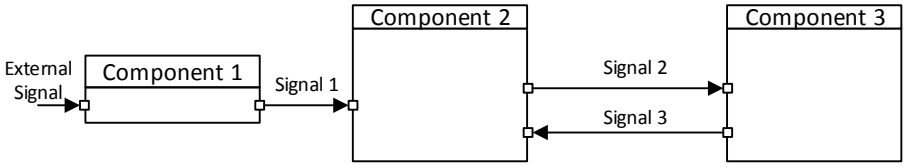
4.1.2 Unified Modeling Language

In the domain of software engineering, the term *model* is inevitably connected to the different diagram types of the Unified Modeling Language (UML), since they are the de-facto standard in both academia and industry to model a software system. UML is the Object Management Group's most frequently used specification, and is described as the *lingua franca* in software engineering for modeling and documentation purposes [MG15, Pet13]. Overall, UML provides fourteen separate diagram types that can be used to describe a system from different perspectives (e.g., behavior, structure) or abstraction levels (e.g., design, analysis). In addition, UML has influenced the development of other modeling languages, such as SysML which focuses on systems engineering. UML assists developers in dealing with complexity and distributes responsibilities among individual stakeholders [Pen03]. The diagrams help to support many software development activities, such as: transforming an analysis model into a design model, transforming a design model into an implementation, generating documentation, model-driven testing, validation and verification, schedulability analysis, and performance estimation [TLG+16]. The following paragraphs introduce the three UML diagram types of interest for this thesis: *activity* diagrams, *block-based* and *state machine* diagrams.

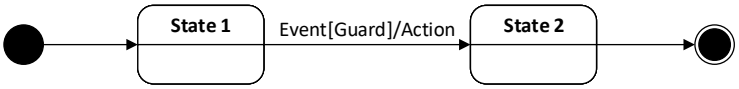
Activity Diagram. An UML activity diagram is a graphical representation of a workflow. Its intended use is to model organizational or computational processes. Fig. 4.1a depicts a simple activity diagram containing all the required elements for our proposed approach. The notation is straightforward with black-colored *initial* and *final* nodes. *Activity* nodes describe the tasks that have to be executed, the diamonds represent *decisions* in the workflow and are enriched with a condition (if-else construct) that has to be fulfilled in order to traverse a specific path. The identical notation is used for *merge* constructs. Finally, activity diagrams support concurrency indicated by the black bars (*fork* and *join* elements). In Fig. 4.1a an object enters the workflow at *Activity 1* and goes to *Activity 2* if the condition



(a) UML Activity Diagram



(b) Block-Based Diagram



(c) UML State Machine Diagram

Figure 4.1: Relevant Diagram Types of UML.

is evaluated to true or to *Activity 3* otherwise. After *Activity 2* is performed the workflow is split into two parallel tasks, i.e. *Activity 4* and *5*, that are simultaneously executed. Afterwards, the workflow joins together before reaching the merge node. Considering the second path, an object simply enters *Activity 6* before entering the merge node. The workflow is finished in the final node. Objects are referred to as workpieces and customers throughout this thesis. Further diagram elements, e.g. swim lanes, are omitted for this thesis as they are not required. Activity diagrams are part of the behavioral system descriptions provided by the UML.

Block-Based Diagram. The second relevant diagram does not directly originate from the UML, since it is not one of the 14 available types. Fig. 4.1b shows a graphical representation of the block-based diagram. It contains *components*, *ports* and *connections*. Ports are the interfaces with which components communicate with each other via directed connections. A connection is equipped with a *signal* that is transmitted following the specified direction. In Fig. 4.1b, we can determine

that *Component 1* receives external input such as temperature values or humidity. Hence, it may reflect a sensor component. The information is transmitted to *Component 2*, which itself communicates with *Component 3* in a bidirectional way. We can see a strong correlation to UML composite structure and UML component diagrams. However, composite structure diagrams are intended to depict the internal structure of classes, e.g. as used in programming language, which are not the focus of this thesis. Concerning component diagrams, connections are realized with interfaces representing a customer-service provider relationship. One component requires data, which is provided by another component. We can imagine to also model signals between different hardware components with this notation for an automation system. In addition, recent studies indicate that UML is applicable as a general purpose architecture description language (ADL) [Pan10, Oqu06, KS00]. In the end, our decision for this special ADL is based on previous work. The block-based diagrams were already successfully used and evaluated to model software architectures, derive test case selection strategies and manage architectural variability [LLL+13, HKR+11a]. They belong to the structural diagrams. Hierarchical components are not considered in this thesis.

State Machines. UML state machines [Har87] are an enhanced realization of the mathematical concept of a finite automaton. Computer science has countless application scenarios for state machines. They are one of the most used diagram types in software development, probably next to UML class diagrams [TLG+16]. Fig. 4.1c depicts an exemplary state machine. The notation is similar to activity diagrams with identical *initial* and *final* states. *Transitions* connect the individual *states* and are by default spontaneous (or internally triggered), unless a triggering event, a guard and a resulting action are specified. Thus, the system in Fig. 4.1c enters *State 1* and can only continue if the *[Guard]* evaluates to true and the defined *Event* occurs. In this case, the system performs the specified *Action* and enters *State 2* before it reaches the end. Analogue to activity diagrams, state machines describe behavior in a software system. Other concepts such as parallel and nested states are not considered in this thesis as the PPU does not require them. However, an extension of the proposed contribution to these elements is imaginable.

4.2 The Modeling Perspectives

An application of MDD principles to the automation domain also involves the consideration of multiple disciplines with mechanical, electrical and software engineering (cf. Chapter 3). We must provide models that are easy to understand and can represent all relevant aspects of an automation system [VHMK+15]. Following the ideas of the SPES XT project where an embedded system is developed in a model-driven

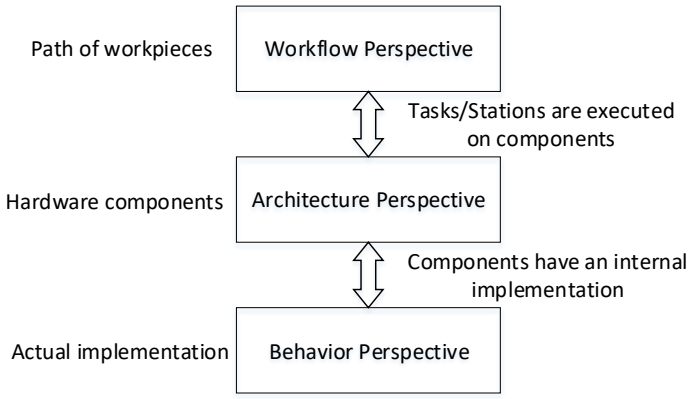


Figure 4.2: Multi-Perspective Design: Overview [KLL+14].

fashion with different interconnected viewpoints [PHAB12], we decided to use three modeling perspectives. Each modeling perspective is represented by one of the previously described UML diagrams, since they are easy to understand for developers of each discipline. The three distinct variants of the PPU introduced in Chapter 2, i.e., **Basic**, **Stamp** and **Optimized**, serve as application scenarios in this section.

4.2.1 Conceptual Design

During the early development stages, it is helpful to visualize the conceptual idea of the automation system. The concept is then refined into a system architecture. This architecture is quite similar to a pure software system architecture in which we have to identify components and their interfaces to exchange data. Nevertheless, automation systems consist of real hardware elements, e.g. robots, sensors, which have to be modeled as well as their communication channels. In a last step, we have to model the behavior of the components based on the previously defined architecture. The complexity is increasing during these modeling steps, since we have to deal with a rising number of details, which we have to consider and therefore model. For example, even one architectural component may contain a state machine with a dozen or more states and each state may have multiple transitions with events, guards or actions.

Our multi-perspective design-level modeling approach consists of a *workflow, architecture* and *behavioral* perspective to model an automation system. Fig. 4.2 shows the hierarchy of the modeling perspectives and their connections. The three modeling perspectives can cover different stages in system development and are capable to include all relevant information of an automation systen. Additionally, elements in any modeling perspective can be mapped to elements in another modeling perspective to create a consistent model of the system. This structure is based on a

typical software development process in which the general functionality and idea of the desired system are first defined in an abstract way, i.e. in form of use cases. The sequence of a use case can then be modeled as UML activity diagram. Afterwards, the development process enters several design phases. First, we create a high level technical design in form of an architecture, which is the basis for a segmentation into a more detailed design such as classes. The implementation effort for software developers can be reduced significantly if the design phases are executed thoroughly [FR07, Pet13]. Naturally, there is also a connection between elements of the different design phases. Thus, we follow a similar process with our modeling approach for automation systems. As discussed earlier, we can use UML component diagrams as alternative representation for the architecture modeling perspective. Although the automation community commonly uses languages from the IEC 61131-3, e.g. ladder and function block diagrams or structured text, to realize their systems, recent observations show a trend to concepts and diagrams of the UML [WVH04, Fri09, VHBKF11]. This aspect encourages our decision to use UML diagrams to model an automation system.

The **Basic** variant of the PPU serves as example demonstrating the concept and realization of each modeling perspective.

Workflow Perspective. The workflow perspective is the most abstract modeling level. It captures the technical process realized by an automation system and describes the path of a workpiece through the automation system as well as the different tasks that need to be performed in the prescribed order (cf. Fig. 4.2). The workflow perspective is described by UML activity diagrams. We refer to the activity nodes as tasks/stations throughout this thesis. Listing 4.1 and Fig. 4.3 show the workflow of the **Basic** PPU variant on this top-level perspective. It is a sequential order of three tasks, since the workpieces are just transported from the *Stack* to the *Slide* using the *Crane*. The structure in Listing 4.1 is fundamentally different compared to Fig. 4.3. At first, we define the name of the model. Then, tasks and control flows are defined separately. The task definition includes a label and a certain type. Labels can be used in case of a transformation to a graphical representation as in Fig. 4.3. The types are identical to the different nodes in activity diagrams. Control flows are defined by giving a source and target task. Developers can choose between textual and graphical development based on their own preferences or the task at hand. While modeling an activity diagram is not challenging in general, it gets increasingly more difficult with the introduction of DM, especially in case of the graphical representation (cf. Section 4.3.1).

```
ActivityDiagram PPU:Basic {
    Tasks{
        Start ( type= Initial, label= "Start" )
```

```
        Stack ( type= Action,  label= "Stack" )
        Crane ( type= Action,  label= "Crane" )
        Slide ( type= Action,  label= "Slide" )
        End   ( type= Final,   label= "End"   )
    }
    ControlFlows{
        flow0 = Start => Stack
        flow1 = Stack => Crane
        flow2 = Crane => Slide
        flow3 = Slide => End
    }
}
```

Listing 4.1: Workflow of the PPU: Textual.

Figure 4.3: Workflow of the PPU: Graphical [KLL+14].

The *Stack* and *Crane* tasks are further refined in the next modeling perspectives with components, their dependencies and finally an internal behavior. The *Slide* is a pure mechanical piece with no communication channels to other components and no implementation, which is why an additional refinement is not necessary.

Architecture Perspective. The architecture perspective captures the logical layout of the automation system, i.e., the technical system, where a component in the architecture represents an actual hardware part, e.g., sensors and actuators. The communication is represented by signals passed between components via ports and connectors making a flow of information possible. Signals can be sent according to the specified direction through ports and connectors. The architecture perspective is described by the block-based diagrams taken from [Lac17, LLL+13].

Fig. 4.4 shows the architecture for the **Basic** variant. The PPU has two main components *Stack* and *Crane*. An extra component for the *Slide* is not necessary, since it is a mechanical slide without any behavior. All remaining components are sensors providing input signals for the *Stack* or *Crane*. They provide the information if there is a workpiece present *wpPresent* at the *Stack*, what the current position of the *Crane atPlace* is or what the current separator state *spRetract* or *spExtract* is at the *Stack*. The textual representation is omitted for this modeling perspective and future parts, since it does not contain any additional information compared to the graphical one. Textual examples for all modeling perspectives are available in [Lor14].

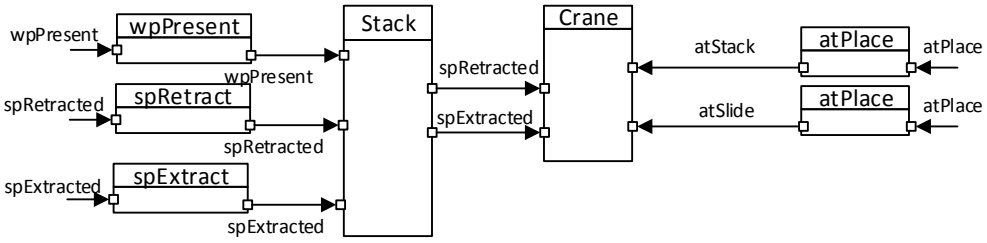


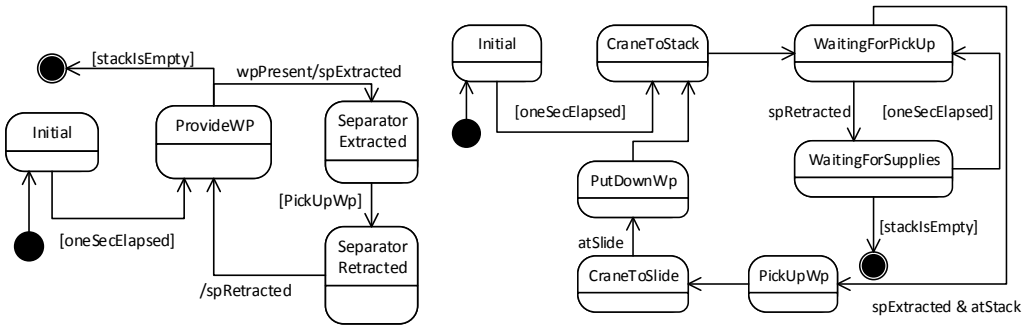
Figure 4.4: Architecture of the PPU: Graphical [KLL+14].

Tasks in the workflow perspective are mapped to components in the architecture perspective meaning that the component is executing this task. A task can be assigned to several components, e.g., the task is executed on multiple machines. Vice versa, a machine can execute several tasks. Our modeling approach facilitates fine-grained and configurable allocation of task execution.

Behavior Perspective. The behavior perspective describes how the single components in the system operate. It is described by UML state machines. Thus, developers can describe the internal *Stack* or *Crane* implementation. Each state machine is assigned to the components on the architectural perspective. Several components can be assigned the same behavior, while it is impossible that the same component has two behaviors at the same time, otherwise it would be another variant of the system. The behavior perspective uses signals introduced in the architecture perspective as triggering events or guards at transitions, but may also use further, implementation-oriented events.

There are two different main components, the *Stack* and the *Crane*, for which we provide behavioral specifications in Fig 4.5. In the state machine for the *Stack*, after an initialization phase, the *Stack* goes into the *ProvideWP* state. For a filled *Stack*, *wpPresent* results in a continuous loop of the *Stack's* pneumatic cylinder separating workpieces from the *Stack* for the *Crane* until the *Stack* is empty. A new workpiece can only be provided after the previous one is picked up *PickUpWp* by the vacuum gripper of the *Crane*. The second component, the *Crane*, needs a different behavioral model. First, the *Crane* needs to move to the *Stack CraneToStack*. The *Crane* uses the signals *spRetracted* and *spExtracted* from the *Stack* about the pneumatic cylinder status. The *Crane* either waits for the next workpiece or picks an available workpiece up. An empty *Stack* results in a final state, whereas an actual workpiece triggers the *Crane* to move to the slide *CraneToSlide*, put down the workpiece and move back to the *Stack*.

In summary, the behavioral perspective permits to specify details in a high-level manner, while still allowing effective code generation, as shown in the implementation part (cf. Section 4.3.1).

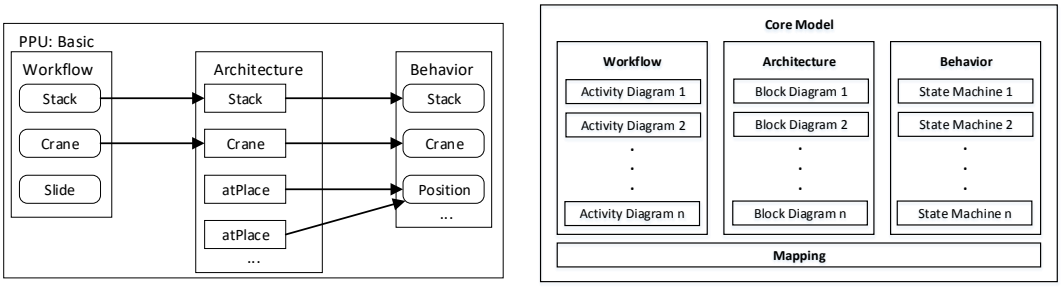


(a) Stack Behavior (b) Crane Behavior

Figure 4.5: Behavior of the main components: Graphical [KLL+14].

Mapping of the Modeling Perspectives. Fig. 4.6a depicts the mapping between different modeling perspectives for the **Basic** variant of the PPU in our graphical realization. Developers get a list of available elements in the respective modeling perspectives and can simply draw connections between them. For instance, the task *Stack* is connected to the component *Stack* in the architecture and obviously to a state machine *Stack* in the behavior modeling perspective. A similar process applies for the *Crane*. As previously stated, the *Slide* does not have any elements besides the task itself and thus it has no connections. The sensor *atPlace* occurs multiple times in the architecture, but the implementation stays identical which is why both components are linked to the same state machine, namely *Position*. Additional components and state machines are neglected at this point, since the general mapping principle remains the same.

The general concept is depicted in Fig. 4.6b. We can have an arbitrary number of models on each modeling perspective. This is especially important in case of the behavior, since each component in the architecture typically has its own state



(a) **Basic** Mapping (b) Core Model

Figure 4.6: Mapping and general structure [KLL+14].

machine. A restriction to the other perspectives in terms of diagram numbers was deemed unnecessary, since developers can always decide for themselves how many separate workflows or architectures they need in order to model a system. The mapping between the individual perspectives is modeled separately (cf. Fig 4.6b, bottom). All of these models make up the *Core* variant of a system.

However, with the *Core*, i.e. **Basic**, we have implemented just one variant of the PPU. In order to create reusable models for the complete PPU product line, we introduce the concept of DM withing all of our modeling perspectives as a next step [KLL+14, KPST14].

4.2.2 Application of Delta Modeling

The main idea of delta modeling (DM) is to represent system variants by explicitly modeling changes to a designated core system. Thus, we can consider the **Basic** PPU variant as our core. A delta contains a set of operations to add, remove or modify model elements. By applying the necessary deltas in an appropriate order to the core system, the desired variant can be automatically generated (cf. Section 2.1.3). The underlying formal semantics can be found in [Sch10]. We instantiate this generic semantics for each modeling perspective in order to model additional variants of the PPU with deltas. In the following, we use the approach to generate the **Stamp** variant of the PPU.

Workflow Deltas. An instantiation of DM, its operations and application scenarios encourages the definition of a solid formalism in advance to simplify the actual realization. Thus, we provide a formal definition of our workflow based on UML activity diagrams at first.

Definition 4.1: Workflow

A workflow is a tuple $Wf = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ where:

- $\mathcal{V}$ is the set of all nodes;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}$ is the set of labeled edges;
- $\mathcal{L}$ is the set of all labels.

This definition specifies a directed graph. Without loss of generality, $\mathcal{V}$ contains the node types: $\mathcal{V} = \Sigma_{Action} \cup \Sigma_{Initial} \cup \Sigma_{Final} \cup \Sigma_{Decision} \cup \Sigma_{Merge} \cup \Sigma_{Fork} \cup \Sigma_{Join}$. Each edge $e \in \mathcal{E}$ has a label $l \in \mathcal{L}$ specifying conditions for traversing the edge, if applicable. Spontaneous transitions between nodes are allowed, since conditions are not always necessary.

In our workflow model, we allow a delta to add and remove nodes of all types. Furthermore, deltas can add and remove transitions. In this way, the control flow can be completely altered. Additionally, transitions can be relabeled. Relabeling

can either be expressed through replacing the existing transition with an add and remove operation or by modifying the transition directly. We now formally define all possible atomic delta operations on our workflow.

Definition 4.2: Workflow Delta Operations

A workflow delta is a set of delta operations $\delta \subseteq WfOp$ for a given set of $Wf = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, where

$$\begin{aligned} WfOp = &\{\mathsf{add}\ v \mid v \in \mathcal{V}\} \cup \{\mathsf{add}\ (v_s, l_{st}, v_t) \mid v_s, v_t \in \mathcal{V}, l_{st} \in \mathcal{L}\} \\ &\cup \{\mathsf{rem}\ v \mid v \in \mathcal{V}\} \cup \{\mathsf{rem}\ e \mid e \in \mathcal{E}\} \\ &\cup \{\mathsf{mod}\ (v_s, l_{st}, v_t)\ \mathsf{by}\ k_{st} \mid (v_s, k_{st}, v_t) \in \mathcal{E}, k_{st} \in \mathcal{L}\} \end{aligned}$$

in which v_s is the source node and v_t the target node of an edge.

A single delta is able to generate a new workflow. However, previous work has shown that a set of deltas can be combined into a single composite delta by defining an appropriate delta composition operation. In order to ensure that the application of such a delta leads to a well-formed workflow, we require that the delta is applicable and consistent. A delta is *applicable* to the workflow if all elements (node, edge or label) which should be removed or modified exist and if all elements which are added do not exist. A delta is *consistent* if it adds, removes or modifies each element at most once [Sch10]. Furthermore, a consistent delta ensures that there are no dangling edges in the resulting workflow. This means that removing a node also causes the removal of all resulting edges. Edges are never added between nodes that are removed in the delta. If a node of an added edge does not exist in the core workflow model, the necessary source and/or target edges are also added in the delta. Ensuring consistency plays a major role in our multi-perspective design level modeling approach, which is why we dedicate a complete subsection to the introduction of a concrete consistency checking concept (cf. Section 4.2.3).

The following definition formalizes how to obtain a valid variant by applying a delta to a workflow.

Definition 4.3: Workflow Delta Application

The application of an applicable and consistent delta $\delta \subseteq WfOp$ to a $Wf = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ is defined by the function $Wf' = apply(Wf, \delta)$, where $Wf' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$. It is recursively defined as follows:

1. Case: $\delta = \emptyset$: $Wf' = Wf$.
2. Case: $\delta = \delta' \cup \delta'' \wedge \delta', \delta'' \in WfOp$: $Wf' = apply(apply(Wf, \delta'), \delta'')$.
3. Case: $\delta = \mathsf{add}\ v$: $\mathcal{V}' = \mathcal{V} \cup \{v\}$
4. Case: $\delta = \mathsf{add}\ (v_s, l_{st}, v_t)$: $\mathcal{E}' = \mathcal{E} \cup \{(v_s, l_{st}, v_t)\}$.

5. Case: $\delta = \textsf{rem}\ v$: $\mathcal{V}' = \mathcal{V} \setminus \{v\}$.
6. Case: $\delta = \textsf{rem}\ e$: $\mathcal{E}' = \mathcal{E} \setminus \{e\}$.
7. Case: $\delta = \textsf{mod}\ (v_s, l_{st}, v_t)\ \textsf{by}\ k_{st}$: $\mathcal{E}' = \Big(\mathcal{E} \setminus \{(v_s, l_{st}, v_t)\}\Big) \cup \{(v_s, k_{st}, v_t)\}$.

Given this formalism, we can define a respective workflow delta for the PPU that contains the necessary changes to the **Basic** variant in order to generate the new **Stamp** variant (cf. Example 4.1). Fig. 4.7 depicts the final result after the delta is applied to the core within our graphical representation of the workflow. All elements from the **Basic** variant are grayed out, while elements changed in the delta are colored in deep black. Listing 4.2 depicts the identical delta within our textual approach. The delta can be addressed by its name *Stamp*. This is followed by the addition of all required nodes. Again, nodes are specified with labels and types. We can remove control flows from any variant by writing *Variant.DesiredControlFlow*. The notation is similar for the removal of nodes which is not necessary in this case. Overall, we are able to access all modeled variants and deltas with this notation. New control flows are added by specifying a source and target task. Furthermore, it is possible to annotate them, e.g. with guards visible for *flow5* and *6*.

Example 4.1: PPU Workflow Delta

δ-Operation from **Basic** $\Rightarrow$ **Stamp**:

$$\delta_{Wf} = \{add\ Crane_M, add\ Stamp, add\ DN, add\ MN, rem(Stack, \epsilon, Crane),$$
$$rem(Crane, \epsilon, Slide),\ add(Stack, \epsilon, DN),\ add(DN, wpBlack, Crane)$$
$$add(DN, wpMetallic, Crane_M),\ add(Crane_M, \epsilon, Stamp),$$
$$add(Stamp, \epsilon, MN),\ add(Crane, \epsilon, MN),\ add(MN, \epsilon, Slide)\}$$

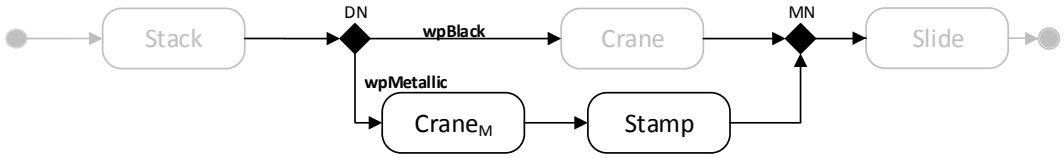


Figure 4.7: Workflow: **Stamp** Variant [KLL+14, KPST14].

```
DeltaActivityDiagram Basic to Stamp {
Stamp {
    addNode Crane_M ( type= Action,   label= "Crane_M" )
    addNode Stamp   ( type= Action,   label= "Stamp"   )
    addNode DN      ( type= Decision, label= "DN"      )
    addNode MN      ( type= Merge,    label= "MN"      )
    removeControlFlow Basic.flow1
```

```
    removeControlFlow Basic.flow2

    addControlFlow flow4 = Basic.Stack => DN
    addControlFlow flow5 = DN => Basic.Crane(guard="wpBlack")
    addControlFlow flow6 = DN => Crane_M(guard="wpMetallic")
    addControlFlow flow7 = Crane_M => Stamp
    addControlFlow flow8 = Stamp => MN
    addControlFlow flow9 = Basic.Crane => MN
    addControlFlow flow10 = MN => Basic.Slide
    }
}
```

Listing 4.2: Workflow Delta: Textual.

Architecture Deltas. The instantiation of DM can be repeated for the block-based diagrams. First, we provide a formal definition for our architecture models.

Definition 4.4: Architecture

An architecture is a tuple $Arc = (\mathcal{C}, \mathcal{P}, \mathcal{CON}, \mathcal{S}, m)$ where:

- $\mathcal{C}$ is the set of all components;
- $\mathcal{P}$ is the set of all ports;
- $\mathcal{S}$ is the set of all signals;
- $\mathcal{CON} \subseteq \mathcal{P} \times \mathcal{S} \times \mathcal{P}$ is the set of connections labeled with signals,
- $m : \mathcal{P} \to \mathcal{C}$ is a function mapping each port to a component.

The central elements in the architecture perspective are components, ports and connections with associated signals. Deltas are able to add and remove components, ports and connections in order to change the architectural structure. Signals are always bound to a connector. Hence, a delta can also change the signal attached to a connector. Thus, we obtain the following formal delta operations.

Definition 4.5: Architecture Delta Operations

An architecture delta is a set of delta operations $\delta \subseteq ArcOp$, where

$$\begin{aligned} ArcOp = {} & \{\mathsf{add}\ c \mid c \in \mathcal{C}\} \cup \{\mathsf{add}\ (p_s, s_{st}, p_t) \mid p_s, p_t \in \mathcal{P}, s_{st} \in \mathcal{S}\} \\ & \cup \{\mathsf{rem}\ c \mid c \in \mathcal{C}\} \cup \{\mathsf{rem}\ con \mid con \in \mathcal{CON}\} \\ & \cup \{\mathsf{mod}\ (p_s, s_{st}, p_t)\ \mathsf{by}\ k_{st} \mid (p_s, k_{st}, p_t) \in \mathcal{CON}, k_{st} \in \mathcal{S}\}. \\ & \cup \{\mathsf{add}\ (p, c_t) \mid p \in \mathcal{P}, m : p \to c_t\} \cup \{\mathsf{rem}\ p \mid p \in \mathcal{P}\} \end{aligned}$$

in which p_s is the source port, p_t the target port of a connection and c_t the target component.

Again, we need an applicable and consistent delta to ensure a well-formed architecture. Applicability at this perspective forces identical properties as in the workflow, i.e., existence of removed and modified elements as well non-existence of elements that should be added to the architecture. The delta is consistent if no component is isolated meaning without any connection to other components. Furthermore, all ports are connected to another port and transmit a signal. Definition 4.6 specifies the delta application in order to generate a new variant.

Definition 4.6: Architecture Delta Application

The application of an applicable and consistent delta $\delta \subseteq ArcOp$ to a $Arc = (\mathcal{C}, \mathcal{P}, \mathcal{CON}, \mathcal{S}, m)$ is defined by the function $Arc' = apply(Arc, \delta)$, where $Arc' = (\mathcal{C}', \mathcal{P}', \mathcal{CON}', \mathcal{S}', m')$. It is recursively defined as follows.

1. Case: $\delta = \emptyset$: $Arc' = Arc$.
2. Case: $\delta = \delta' \cup \delta'' \wedge \delta', \delta'' \in ArcOp$: $Arc' = apply(apply(Arc, \delta'), \delta'')$.
3. Case: $\delta = \mathsf{add}\ c$: $\mathcal{C}' = \mathcal{C} \cup \{c\}$.
4. Case: $\delta = \mathsf{add}\ (p_s, s_{st}, p_t)$: $\mathcal{CON}' = \mathcal{CON} \cup \{(p_s, s_{st}, p_t)\}$.
5. Case: $\delta = \mathsf{rem}\ c$: $\mathcal{C}' = \mathcal{C} \setminus \{c\}$.
6. Case: $\delta = \mathsf{rem}\ con$: $\mathcal{CON}' = \mathcal{CON} \setminus \{con\}$.
7. Case: $\delta = \mathsf{mod}\ (p_s, s_{st}, p_t)\ \mathsf{by}\ k_{st}$: $\mathcal{CON}' = \Big(\mathcal{CON} \setminus \{(p_s, s_{st}, p_t)\}\Big) \cup \{(p_s, k_{st}, p_t)\}$.
8. Case: $\delta = \mathsf{add}\ (p, c_t)$: $\mathcal{P}' = \mathcal{P} \cup \{p\} \wedge m' \mid m : p \rightarrow c_t$.
9. Case: $\delta = \mathsf{rem}\ p$: $\mathcal{P}' = \mathcal{P} \setminus \{p\}$.

Finally, we are able to define a delta for our architecture and the **Stamp** variant of the PPU. We only show an extract of the complete delta based on the formal notation in the following example:

Example 4.2: PPU Architecture Delta

Extract of the δ-Operation from **Basic** $\Rightarrow$ **Stamp**:

$$\delta_{Arc} = \{add\ wpBlack, add(p_{in}, wpBlack), add(p_{out}, wpBlack)$$
$$add(\epsilon, wpBlack, p_{in}), add(p_{out}, wpBlack, p_{Stack}), ...\}$$

This delta fraction adds the component *wpBlack* and its dependencies to the architecture. Other components can be added likewise.

The final result after the application of the complete delta is depicted in Fig. 4.8. Again, the deep black parts indicate all executed changes to the core model. A *Stamp* component has to be added to the architectural perspective with several sensors for the position of the workpiece within the *Stamp* component. The position is signaled to the *Crane*, which provides the *Stamp* module with metallic workpieces.

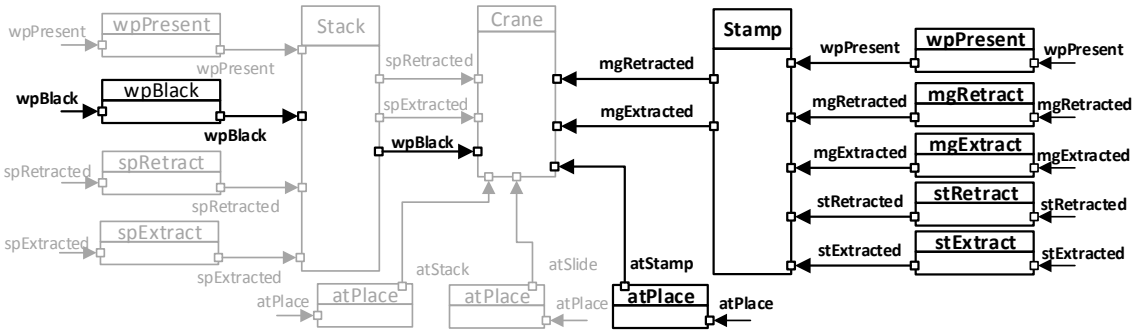


Figure 4.8: Architecture: **Stamp** Variant [KLL+14, KPST14].

Therefore, a new position sensor is added to the *Crane* as well. The identification of the workpiece type is done at the *Stack* with an additional sensor called *wpBlack*.

Behavior Deltas. The instantiation of DM for state machines is similar to activity diagrams, since both diagram types are a directed graph with nodes/states, edges/-transitions and labels. Thus, we define the formalism for state machines in analogy to our workflow:

Definition 4.7: State Machine

A state machine is a tuple $SM = (\mathcal{S}, \mathcal{T}, \mathcal{L})$ where:

- $\mathcal{S}$ is the set of all states;
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the set of labeled transitions;
- $\mathcal{L}$ is the set of all transition labels.

Again, this definition specifies a directed graph. Without loss of generality, $\mathcal{S}$ contains the state types: $\mathcal{S} = s_0 \cup \Sigma_{Final} \cup \Sigma_{State}$ where s_0 is the initial state. Each transition $t \in \mathcal{T}$ has a label $l \in \mathcal{L}$ specifying conditions for traversing the transitions. In this case, the set of labels is a conjunction of $\mathcal{L} = \Sigma_{Events} \cup \Sigma_{Guards} \cup \Sigma_{Actions}$. Spontaneous transitions between states are allowed.

State machines contain states and transitions, which can be changed with deltas. A delta can add and remove states and transitions in order to change the modeled behavior. Deltas can also change the transition labels consisting of an event, a guard and an action in any combination. Thus, we can formally define all atomic delta operations as follows:

Definition 4.8: State Machine Delta Operations

A state machine delta is a set of delta operations $\delta \subseteq SMOp$ for a given set of $SM = (\mathcal{S}, \mathcal{T}, \mathcal{L})$, where

$$
\begin{aligned}
SMOp = \{&\mathsf{add}\ s \mid s \in \mathcal{S}\} \cup \{\mathsf{add}\ (s_s, l_{st}, s_t) \mid s_s, s_t \in \mathcal{S}, l_{st} \in \mathcal{L}\} \\
&\cup \{\mathsf{rem}\ s \mid s \in \mathcal{S}\} \cup \{\mathsf{rem}\ t \mid t \in \mathcal{T}\} \\
&\cup \{\mathsf{mod}\ (s_s, l_{st}, s_t)\ \mathsf{by}\ k_{st} \mid (s_s, k_{st}, s_t) \in \mathcal{T}, k_{st} \in \mathcal{L}\}
\end{aligned}
$$

in which s_s is the source state and s_t the target state of a transition.

Again, we need an applicable and consistent delta to ensure a well-formed state machine. Applicability at this modeling perspective also forces properties as in the workflow, i.e., existence of removed and modified elements as well non-existence of elements that should be added to the state machine. The delta is consistent if no state is isolated meaning without any transition to other states. Definition 4.9 formally specifies the delta application in order to generate a new variant in this modeling perspective.

Definition 4.9: State Machine Delta Application

The application of an applicable and consistent delta $\delta \subseteq SMOp$ to a $SM = (\mathcal{S}, \mathcal{T}, \mathcal{L})$ is defined by the function $SM' = apply(SM, \delta)$, where $SM' = (\mathcal{S}', \mathcal{T}', \mathcal{L}')$. It is recursively defined as follows:

1. Case: $\delta = \emptyset$: $SM' = SM$.
2. Case: $\delta = \delta' \cup \delta'' \wedge \delta', \delta'' \in SMOp$: $SM' = apply(apply(SM, \delta'), \delta'')$.
3. Case: $\delta = \mathsf{add}\ s$: $\mathcal{S}' = \mathcal{S} \cup \{s\}$
4. Case: $\delta = \mathsf{add}\ (s_s, l_{st}, s_t)$: $\mathcal{T}' = \mathcal{T} \cup \{(s_s, l_{st}, s_t)\}$.
5. Case: $\delta = \mathsf{rem}\ s$: $\mathcal{S}' = \mathcal{S} \setminus \{s\}$.
6. Case: $\delta = \mathsf{rem}\ t$: $\mathcal{T}' = \mathcal{T} \setminus \{t\}$.
7. Case: $\delta = \mathsf{mod}\ (s_s, l_{st}, s_t)\ \mathsf{by}\ k_{st}$: $\mathcal{T}' = \Big(\mathcal{T} \setminus \{(s_s, l_{st}, s_t)\}\Big) \cup \{(s_s, k_{st}, s_t)\}$.

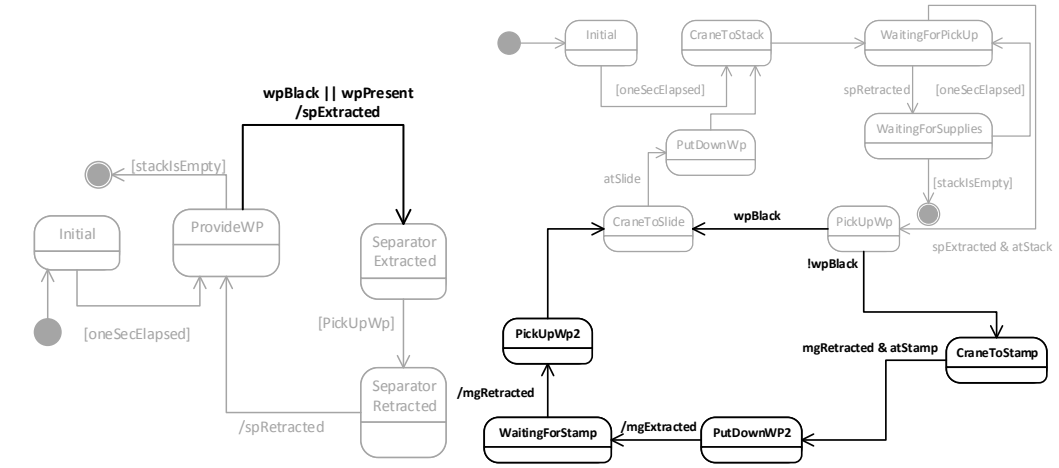
The behavior perspective contains the most changes in order to successfully generate the **Stamp** variant. Fig. 4.9 depicts the final result and an application of the formalism is given in Example 4.3. In this modeling perspective, a new state machine for the *Stamp* is added, which captures a sequential process. Either the *Stamp* is waiting for a new workpiece or the separator puts the piece under the *Stamp*. After the *Stamp* process, the *Crane* transports the workpiece to the *Slide*. The *Stack* is identical to the *Stack* in the **Basic** variant except for the differentiation of the two workpieces in one event. Similar to the workflow, the crane component is extended by an additional path for the stamping process.

Example 4.3: PPU State Machine Delta

Extract of the δ-Operation from **Basic** $\Rightarrow$ **Stamp**

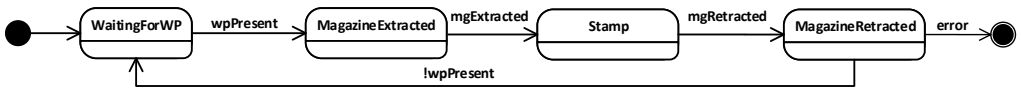
$$\delta_{SM} = \{add\ q_0, add\ WaitingForWP, add\ MagazineExtracted, add\ Stamp,$$
$$add\ MagazineRetracted, add\ finalState, add(q_0, \epsilon, WaitingForWP),$$
$$add(WaitingForWP, wpPresent, MagazineExtracted)\},$$
$$add(MagazineExtracted, mgExtracted, Stamp),$$
$$add(Stamp, mgRetracted, MagazineRetracted),$$
$$add(MagazineRetracted, error, finalState),$$
$$add(MagazineRetracted, !wpPresent, WaitingForWP)$$

This delta part adds the new state machine for the stamp component to the behavior perspective. Changes to the crane and stack state machines can be defined likewise.



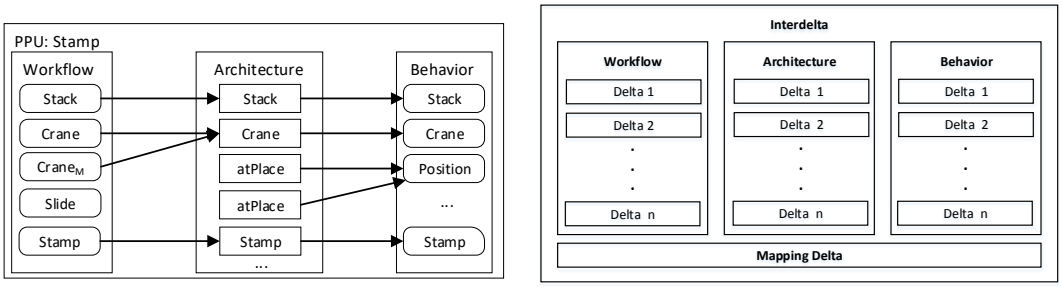
(a) Stack Behavior

(b) Crane Behavior



(c) Stamp Behavior

Figure 4.9: Behavior: **Stamp** Variant [KLL+14, KPST14].



(a) Mapping for the **Stamp** (b) Interdelta

Figure 4.10: Mapping and general delta structure [Lor14, KLL+14].

After the application of a set of deltas in order to generate a variant, it must be ensured that still a valid system variant is achieved. This is required for the single modeling perspectives in isolation, as well as for the mappings between the different perspectives (workflow/architecture, architecture/behavior). These mappings might have to be adjusted after changing a single perspective separately. Therefore, also delta operations exist which add, remove or modify the mapping between the perspectives [KLL+14, KPST14, GÏ5]. Fig. 4.10a shows the final mapping. We can observe that the task $Crane_M$ is also linked to the component *Crane*, because it is the same physical hardware part in reality and its implementation is just changed by a few states. It is also possible to model this variant with just one *Crane* task. However, we need this distinction for our performance analysis in the next chapter. In addition, the *Stamp* has a continuous connection across all modeling perspectives. Further changes are not necessary for the **Stamp** variant.

Besides the textual and graphical representation of the three modeling perspectives, we can define the respective delta operations for each modeling perspective as well as the mapping. Fig. 4.10b indicates that deltas are combined into the so-called *Interdelta* structure. The idea is to model all changes required to generate a valid new variant of the system in one *Interdelta*. For instance, the previously described deltas to enable the stamping process at the PPU would be part of one interdelta. However, a general limitation does not exist in this approach. It is also possible to define multiple interdeltas and apply them in a sequential order to generate a valid new variant.

4.2.3 Consistency Checking Strategies

Using modeling perspectives during the development process ultimately results in a strong dependency between them making consistency an important aspect to ensure a valid system in the end. UML does not provide us with a formal notation of consistency rules at all. Inconsistencies may already occur during the development

of one model. This aspect is intensified by consideration of multiple models for different perspectives [IIS+11]. Inconsistencies can easily lead to errors in the software system [HKRS05, MBC05]. In Fig. 4.9 and more specifically the state machine for the *Stack*, a removal of the transition from state *ProvideWP* to state *Separator Extracted* results in a deadlock that would require a shutdown of the PPU. Hence, it is paramount to identify inconsistencies as early as possible, e.g., during the design phase, and fix them afterwards [SZ01, TLG+16]. The detection of inconsistencies requires a set of consistency rules, which we discuss in the following.

Consistency Rules. Some informal specifications for consistency rules can be found in the UML standard and are referred to as rules for well-formed models [MG15]. Most consistency rules defined in the literature are related to UML class diagrams, which are not present in our multi-perspective modeling approach. In general, the consistency rules found in the UML standard or other literature help us to ensure consistency on each perspective in isolation. For our modeling approach, it is also mandatory to ensure consistency across the three perspectives Thus, we have the following rule categories for our approach:

1. Intra-perspective rules affect each perspective in isolation.
2. Inter-perspective rules affect one full variant of the system.

We identified 22 rules that have to be fulfilled in order to receive a valid workflow [KS16, Kre16]. The rules vary from proving the existence of an initial or final node to reachability checks for each node. An architecture as well as a state machine must comply to 11 different consistency rules such as each port must have a connector or a state machine has exactly one initial state. The difference in numbers can be explained by a significantly larger number of model elements at the workflow level. Hence, we must validate a set of 44 different consistency rules for the intra-perspective category. The mapping between the perspectives, i.e., the inter-perspective, can be validated by using an additional number of 5 rules, e.g., each state machine is connected to at least one component in the architecture. A detailed list of all rules can be found in [Kre16].

Given this set of rules, it is possible to use different validation strategies [TAK+14], whereas each concept has another impact on the actual number of checks performed to enforce validity across the variants. We discuss the application of two product-based strategies (cf. Chapter 2) in the following [KS16].

Product-based Consistency Checking. A product-based technique analyzes all variants of the product line in isolation [TAK+14]. Thus, all consistency checks are executed for each individual variant of the PPU. First of all, the developer has to

select the interdeltas leading to the variants for which a validation of consistency is desired. Next, the complete variants are generated within our multi-perspective modeling framework, e.g. **Stamp** and **Optimized**. **Basic** is the core and does not need to be generated. In a next step, we can start the product-based consistency checking on the three variants. First, we validate our core with the complete set of consistency rules. Afterwards, this process is iteratively repeated for each generated variant. However, since we also re-validate modeling perspectives that may never be touched, we have to execute lots of redundant checks. This is a brute force approach, which is feasible for the small automation product line of the PPU, but becomes increasingly more inefficient for a larger variant space [KS16]. Our actual implementation continuously checks all models of the core during development, which is why we could skip its validation in reality [Kre16].

Product-based Incremental Consistency Checking. Our product-based incremental approach consists of similar steps as the previous one. Again, developers have to select the respective interdeltas and generate the final variants in our multi-perspective modeling framework. And again, we initially validate the core variant ensuring that all perspectives and their mappings are consistent. However, each delta encapsulates the modifications executed on the specified model in the respective modeling perspective. Taking this information into account, we can improve the product-based method in two aspects: First, we do not have to recheck models that are already valid. Most likely a delta does not touch all modeling perspectives, e.g., it only modifies one modeling perspective, making consistency checks on the remaining perspectives obsolete. Second, some changes do not even require another iteration on the modified model, e.g., modifying the performance values in the workflow (cf. Chapter 5). The decision to recheck a specific model always depends on the types of modifications in the delta. Based on the transformations that are possible in deltas and the defined consistency rules, we are able to decide if any rule may be violated by the transformation and, therefore, if a revalidation of the model is necessary. As a result, we can expect a reduction of the consistency checks compared to the previous product-based approach, since we avoid many redundant checks. However, this is only the case in small deltas that are limited to some models. A large delta affecting all models and the mapping would result in a validation of the complete variant, and we would lose the incremental benefits here [KS16].

Overall, the presented and realized consistency checking concepts still leave space for improvement. Family-based and feature-based strategies that incorporate more variability knowledge from deltas or the feature model would certainly provide a greater benefit [TAK+14].

4.3 Case Study

A description of the prototypical implementation marks the beginning of this section. Afterwards, we use the presented concepts to model the **Optimized** variant of the PPU. In particular, we plan to answer the following research questions:

- **RQ 4.1**: *Is our modeling approach expressive enough to represent a real-world automation product line?*
- **RQ 4.2**: *Is it feasible to use the approach during the complete development process?*

4.3.1 Implementation

The multi-perspective design-level modeling approach is fully implemented as both textual and graphical editors. The textual implementation uses domain-specific languages (DSLs) on all three modeling perspectives realized with Xtext[2]. Xtext is an open-source framework for developing DSLs and part of the Eclipse Modeling Framework (EMF). It allows the user to write a grammar, which forms the foundation of the DSL. Listing 4.3 depicts a snippet of the Xtext grammar for our state machines. We can link grammar definitions together via grammar-mixin enabling us to reuse specified signals in the architecture within the state machines (Line 1-3).

```
import "http://www.isf.tubs/Architecturearx" as arx
...
  (variabels+=Variable)*
    'statechart' name=ID '{'
5       ('states' '{' (states+=State)* '}')
        ('alphabet' '{' (labels+=Label)* '}')
        ('transitions' '{' (transitions+=Transition)* '}')
     '}';
State:
10    name=ID;
Transition:
    name=ID '=' from=[State|QualifiedName] '->'
            to=[State|QualifiedName] ':' label=[Label];
...
```

Listing 4.3: Excerpt of the State Machine Grammar.

Furthermore, a state machine has a unique identifier, an arbitrary number of states, transitions and an alphabet containing a set of labels (Line 4-8). States and transitions themselves also have a unique identifier with which they are accessed in the modeling perspectives. In Line 12-13, the transition relation is defined by source

[2] http://www.eclipse.org/Xtext/

and target state as well as a label from the alphabet. Of course, the label itself is either an event, guard, action or any combination. More details about the developed Xtext grammars are available in [Lor14].

Several comfort functions such as syntax highlighting and auto completion are available due to the implementation with Xtext. Each DSL in our tool underlies a grammar that already prevents some basic inconsistencies such as all model elements in one modeling perspective must have a unique name or nodes in the workflow must be of a defined type (action, decision, fork,...). However, Xtext is only capable of checking consistency rules with respect to the underlying grammar. Thus, Xtext cannot determine if a node is unreachable or all specified signals are used at one point. We implemented additional consistency rules and their respective checks directly in the Java source code (cf. Section 4.2.3). Developers are notified about inconsistencies by errors in the Eclipse workspace. We can ensure the consistency of the complete core model during its development. However, during the modeling of deltas we can only rely on build in Xtext mechanics, since our consistency checking concept is currently implemented to operate on final and complete variants. The general workflow in our tool is to first model the core and any number of deltas. Second, developers can generate the desired variants. The application of an interdelta is only possible if the models that are modified actually exist in the respective modeling perspectives. If the interdelta is applicable, we execute the concrete modifications defined in a delta. Afterwards, all variants can be analyzed with a product-based or product-based incremental approach to identify inconsistencies (cf. Section 4.2.3) [Kre16, Lor14].

Graphical Delta Modeling. Delta modeling already exists for a wide variety of languages from pure programming such as Java [SBDT10] to modeling languages, e.g., MontiArc in which also software architectures are modeled [HKR+11a]. All instantiations have in common that deltas are specified only on a textual basis. While this is necessary in a programming language where classes, methods and attributes can be changed by deltas, we identified it as a major drawback for graphical modeling languages. Developers often prefer a graphical development process, if they are confronted with UML diagram, simply because it is the common representation, which is lectured in academia and provided by the OMG [MG15]. Additionally, we may lose some advantages of the MDD process and ultimately have a slower perception and comprehensibility for developers in case of large models. Of course, a state machine with thousands of states is also not comprehensible in a graphical way, and we would need adequate counter-measurements, i.e. hierarchical states. The textual definition of a delta in our concept is straightforward. Listing 4.4 describes a delta operation on the behavior perspective for the stack component of the PPU. The semantics of the delta is identical to the graphical representation previously depicted

in Fig. 4.9. We have two alternative options to change the transition label. The modify-operation simply allows us to directly change the label (Line 5). However, it is also possible to remove the old label as well as the transition and add respective new model elements (Line 8-11).

```
DeltaStatechart Basic to Stamp{

Stack{
    //Short Version
5   modify label of t2_3 to (wpMetallic||wpBlack/spExtracted)

    //Long Version
    remove transition t2
    remove label t2_3
10  add label t2_3b (wpMetallic||wpBlack/spExtracted)
    add transition t2b = ProvideWP->SeparatorExtracted:t2_3b
}}
```

Listing 4.4: Textual Delta Representation.

For graphical DM, we need a new concept in order to distinguish between the core model and all individual delta models. We identified three possible approaches that may fulfill the requirements using *colors*, *symbolics* and *layers* [Mey14, GÏ5]. The *color*-approach gives each delta its own unique color. For instance, changes executed in the first delta are green, changes in the second delta are colored in red and so on. Developers are always confronted with the core and all existing deltas at once. The major drawback is the scalability, since we have to assume dozens of deltas. Even for the small SPL of the PPU with its 15 explicitly defined variants, we would have to distinguish 15 different colors. The number of deltas increases significantly even for the PPU, if we do not consider that one delta generates a new variant, but the new variant is composed of several deltas executed in a sequential order. A similar problem arises with the *symbolic*-approach in which each delta is represented by a unique shape. For instance, the core is depicted by solid lines, the first delta by dashed lines and a second delta by dotted lines. Even if we combine both approaches, the scalability would not improve significantly. Both concepts suffer from the unsolved problem of how we can visualize delta operations on top of the delta itself [Mey14, GÏ5].

Thus, we decided to realize a concept using separate modeling layers for each delta. Fig. 4.11 depicts the basic idea using the stack component of the PPU in two variants. At the first layer (left-side), we model the core of the state machine for the *Stack* as in the **Basic** variant of the PPU (see also Fig. 4.5). In order to model the delta to generate the **Stamp** variant, we create a new modeling layer (cf. Fig. 4.11, right-side) and specify the changes (see also Fig. 4.9). Modeling layers,

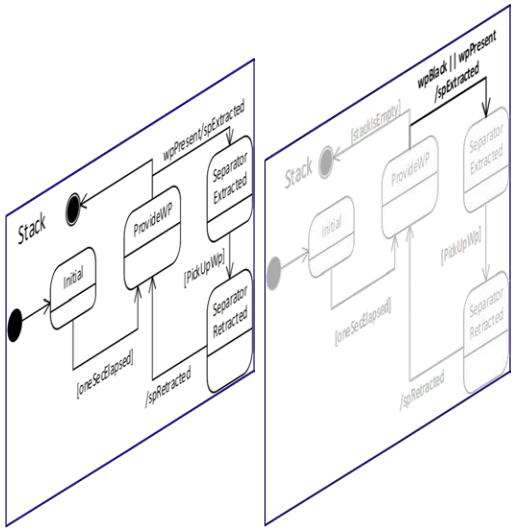


Figure 4.11: Layer Concept

that are not edited at the moment, are shifted into the background. The graphical representation indicates this by graying them out. We are still able to access them from the current layer, since a delta may change all already existing model elements. Developers can select and de-select deltas or introduce nested deltas. A combination with one of the previous approaches enables us to identify additions, removals and modifications in the respective delta, e.g., removals can be symbolized by dashed lines or a red color.

This implementation is realized with the Graphical Modeling Framework (GMF)[3] and the Graphical Editing Framework (GEF)[4]. The decision fell on GEF as it supports maximum customization, which is necessary for our application purposes with delta modeling. We can model each modeling perspective and the mapping with the help of graphical editors. The overall process is identical to the textual realization [GÏ5, Ant15, Mey14].

Variant Generation. Up to this point, we can fully model an automation system such as the PPU within our tool chain. The ultimate goal is to actually control the real-world system with the help of our models which is why we realized a connection to an automation control software. Given a complete variant, core or any generated one, we can export it to CODESYS[5], which is a widely used industrial development tool for automation control software enforcing the IEC 61131 standard. The export format is based on XML generating an almost complete CODESYS project with

[3] `http://www.eclipse.org/gmf-tooling/`
[4] `https://eclipse.org/gef/`
[5] `http://www.codesys.com/`

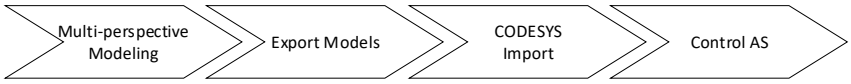


Figure 4.12: Workflow to CODESYS [KLL+14].

the help of Java Emitter Templates[6], which is imported using the CODESYS UML-plugin. In order to provide a meaningful model-to-model transformation, we had to annotate our model elements with positioning values needed by the CODESYS plugin. Otherwise, all model elements would lie at the exact same point, which is not desired by any developer. Therefore, we implemented a heuristic distributing the model elements across the workspace in CODESYS. Fig. 4.12 sketches the workflow. Developers can use our multi-perspective modeling framework to create variants. After the export and import steps, we can observe our models in the CODESYS workspace. Finally, executing the program within CODESYS lets us control the connected automation system. The terminal configuration is not included in the export format and still needs to be provided by CODESYS.

Connecting Spaces. We have to make the solution space UML models configurable from the problem space feature model developed in the first contribution. Thus, we need a connection or mapping between the different spaces and their artifacts that allows users to select a specific feature configuration and the complete variant is automatically generated within our multi-perspective modeling approach. Indeed, we can specify an application condition as well as an application order within our textual editors. Listing 4.5 depicts an exemplary application for a variant of the PPU which adds a conveyor belt. Users can select a concrete product configuration, e.g., *ConveyorLine*, which is derived from a feature model and afterwards the tool automatically selects the deltas with the respective tag and generates the variant. In addition, this delta must be applied *after* the delta for stamping module has been executed. The user is guided through this process by a wizard in Eclipse.

```
Conveyor{
    after Stamp
    when "ConveyorLine"

    addNode ...
    addControlFlow ...
}
```

Listing 4.5: Conditions.

Hence, the missing link between problem and solution space artifacts is also available concluding the implementation part.

[6] https://eclipse.org/modeling/m2t/?project=jet

4.3.2 Feasibility Study

Throughout this chapter, we successfully applied our multi-perspective design level modeling approach to develop a single system with the **Basic** PPU variant and to model variability with the **Stamp** variant. We exemplarily apply the proposed concepts to another variant of the PPU with **Optimized** (cf. Section 2.2) to show the feasibility of our approach. This variant is identical to the **Stamp** at the hardware level. We just optimize the *Crane* implementation in a way that the crane no longer waits at the *Stamp* for the stamping process to be finished. Instead, the *Crane* moves back to the *Stack* to pick up the next plastic workpiece (if present) and transports it to the *Slide*. This change can be interpreted as a software optimization in a later release or can be sold as "premium" behavior.

The step from **Stamp** to **Optimized** causes a performance improvement for the *Crane* (cf. Chapter 5). Assuming that workpieces vary each time, the *Crane* takes the metallic ones to the *Stamp*, returns to the *Stack* in order to pick up the black plastic ones and transports them to the *Slide* before getting the stamped metallic pieces from the *Stamp*. This increases the throughput of the system. As depicted in Fig. 4.13, the delta changes solely the behavioral perspective of the *Crane* component. In its state machine, we add several states after the state *PutDownWP2*. If a metallic workpiece is followed by a non-metallic one, the *Crane* moves back to the *Slide* to transport the non-metallic workpiece to the *Slide* while the metallic workpiece is still stamped. The *Crane* enters the additional loop only if a non-metallic piece is available at the stack. In case of a second metallic workpiece, the *Crane* finishes processing the piece, which is at the *Stamp* [KLL+14].

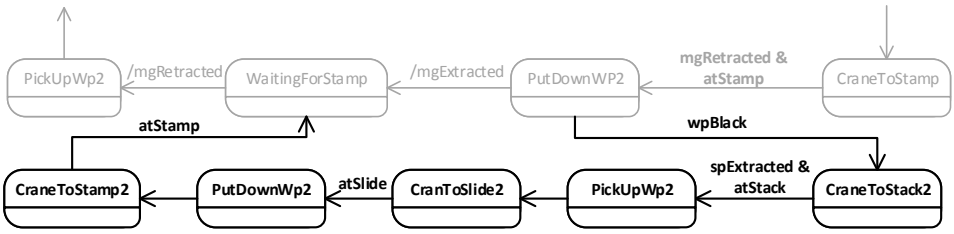


Figure 4.13: Necessary changes to the *Crane* implementation [KLL+14].

We modeled the remaining variants of the PPU likewise. All created models are available in the following student theses [Lor14, Ant15, GÏ5].

Discussion. Next, we summarize our observations in a condensed form regarding the defined research questions.

> **RQ 4.1:** *Is our modeling approach expressive enough to represent a real-world automation product line?*

The application of our modeling approach can be seen as a success. We are able to express all relevant parts of an automation system. In addition, we can model variability through the application of delta modeling. By supporting both, textual and graphical development, developers can select their preferred option. The consistency checking concept mitigates faults during early stages of the development process.

> **RQ 4.2:** *Is it feasible to use the approach during the complete development process?*

Indeed, we are able to support developers from early design phases to the actual execution on a physical machine. A successful application to three variants of the PPU supports this claim.

Threats to Validity. The largest threat lies within the number of case studies. We were only able to test our approach on a single automation system with the size of a lab demonstrator (cf. Section 2.2). A typical automation system such as KeMotion and KePlast is significantly larger in terms of number of hardware and software elements as well as actual size. In addition, the PPU does contain few specified variants for which a product-based consistency checking approach is still viable. Larger product lines would require more advanced concepts for this purpose.

Next, we considered variability in automation system, but neglected the fact of evolution. However, system evolution is a common companion in the domain of long-living systems affecting both hardware and software. We argue that our general concept is still applicable, since delta modeling can capture evolution by the same means [Sch10, LKS16]. In any case, the feasibility of our approach must be re-evaluated for this case.

The feasibility of the presented results is also threatened by the lack of users from other domains. While the approach is feasible enough for software engineers, we have no data concerning the feasibility for a mechanical and electrical engineer. This aspect is especially important, which is why we pursue it further in the future work part of this chapter.

4.4 Related Work

In the following, we separately examine existing work regarding variability in the domains of software and automation system development. First, we focus on related work in software engineering. Second, we review existing work on capturing

variability in industrial plant automation as our intended field of application. The section is concluded by related work on consistency checking in UML diagrams.

Variability Modeling for Software Systems. Variability is studied extensively in the context of SPLs [PBvdL05]. Variability modeling approaches in SPLs can be separated into three different classes: *annotative*, *compositional* and *transformational*. Annotative methods consider one model representing all products of the product line. Variant annotations, e.g., using stereotypes in UML models [Gom06] or presence conditions [CA05], define which parts of the model have to be removed to derive a concrete product variant. Annotative variability models become easily very complex and unmanageable for large SPLs with many variants. Compositional approaches associate model fragments with features, which are composed for a specific feature configuration. A well-known example is AHEAD developed by **Batory et al.** [BSR03], in which a product is incrementally built using a base module and a sequence of feature modules. **Noda et al.** [NK08] construct models by aspect-oriented composition. Other approaches to represent variability on a modeling level, e.g. by **Prehofer** [Pre04] and **Klein et al.** [KPR97], have focused more on composing and adding features and not on capturing evolution. Compositional approaches can only add functionality to an existing product, and the impact of a feature is limited by the used composition technique. In contrast, evolution inevitably needs refactorings, which is very restricted by using compositional methods. Model transformations are, for instance, applied in the common variability language where the variability of a base model is described by rules how modeling elements of the base model have to be substituted in order to obtain a particular product model [HMPO+08]. Delta modeling is a modular transformational approach to design and implement SPLs introduced by **Schaefer** [Sch10]. Delta modeling has so far been applied to represent variability of software architectures [HKR+11a] and Java programs [SBDT10]. Finally, a considerable different approach to handle variability is presented by **Jörges et al.** [JLM+12]. They use a constraint-based method to define valid combinations of artifacts by semantic annotations. Variants are specified in a top-down manner by successively limiting the permitted combinations until a final product is reached.

We use a design-level modeling approach consisting of three modeling perspectives, which provide different viewpoints of the system. Individual developers can focus on different system aspects following the principle of separation of concerns. A similar approach is pursued in the SPES project where an embedded system is developed in a model-driven fashion with different interconnected viewpoints [PHAB12]. The language for the architecture viewpoint is based on work by **Lochau et al.** [LLL+13] in which it is used for delta-oriented testing of large-scale systems.

Variability Modeling in the Automation Domain. Ensuring the changeability of automation systems was lately identified as one major challenge for future industrial competitiveness [LLVH13]. Model-based software and systems engineering is intensively studied over the last decades, e.g., [Vya13, YVP13, BSBF11, TPK07]. An UML model to provide support for re-engineering of automation control software by automatically identifying operation sequences for the control software is presented by **Legat et al.** [LSVH13]. This approach might be integrated as extension of the architecture perspective for (semi-)automatically deriving the state machines. **Colla et al.** [CL11] use event-based models to generate control code for cyclically executed controller platforms (as typically used in automation systems) with some performance limitations. In [WVH11], **Witsch et al.** propose a specialized UML state machine for cyclic control behavior of automation control software. Within our modeling approach, such platform-specific execution behavior is abstracted as an event-based behavior. A lot of research has also been conducted on dynamic reconfiguration of control software, i.e., changing the control behavior during operation [KMLH11, VM10]. This aspect is especially interesting regarding a point left for future work in this thesis with runtime adaptation and delta models at runtime (cf. Section 4.5). **Froschauer et al.** [FDG08] use a design time variability model based on SysML for modeling variability of function blocks. This information is later used to automatically reconfigure (event-based) IEC 61499 control software. Based on the same standard, **Suender et al.** [SVZ11, RSS+07] propose an approach for downtimeless changeability of control systems. A mapping between event-based and cyclically executed control platforms is already possible [CL11]. Thus, we can transform our event-based state machines into any control software that has a cyclic execution time. Nevertheless, **Legat et al.** [LLVH13] concluded that for automation systems, the dependencies of knowledge generation during engineering, i.e., modeling, and knowledge application for adapting the automation systems is still an open issue in industrial practice. The approach presented within this thesis can be seen as a first step towards closing this gap.

Consistency Checking. UML is often the de-facto standard in industry to develop software systems and simultaneously model-to-code generators get more and more popular as they reduce development efforts [UNThEs08]. The correctness of generated code heavily depends on consistency in the UML models which is why consistency techniques are a key aspect in MDD [UNThEs08, TAK+14]. **Usmann et al.** [UNThEs08] identified in their survey five consistency types in UML models with inter-model, intra-model, evolution, semantic and syntactic consistency. The types are based on a literature review concerning existing UML consistency checking methods. Most of the methods are based on class-, sequence- and state machine models leaving our architecture and workflow perspectives excluded. Nonetheless,

we adapted the general consistency types in our work. A more recent study supports the results of the survey [TLG14]. **Mens et al.** [MSS05] also proposed a classification of inconsistencies in their work.

Egyed et al. have done an extensive amount of research to ensure consistency in UML models for systems without variability. In [Egy06], an instant consistency checking method working on the three most popular UML models (see previous paragraph) has been proposed and implemented in a tool called UML/Analyzer, which is integrated within the famous IBM Rational Rose^{TM}. Similar to this work, our core is also instantly validated during development. However, this is not possible for the variability, which is not considered in [Egy06]. Several other approaches exist that can execute consistency checking in UML models for single software systems [UNThEs08, LHE10, Egy07]. In all cases, the main limitation of these approaches is that they do not treat system variability. Managing inconsistencies in systems with variability and evolution requires an incremental consistency checking method, which can be based, e.g., on delta modeling. The incremental nature is mandatory due to the large number of variants that we may have to check and to reduce the response time for the developers. Identifying inconsistencies is often not sufficient enough. One has to aim at fixing the detected inconsistencies in the future (cf. Section 4.5). Some of the proposed techniques in the literature already include methods for fixing UML models [Egy07, RE12], but again without considering variability. Repairing variant-rich systems is a much more challenging task. In the research-in-progress paper by **Lopez-Herrejon et al.** [LHE12], a preliminary study how to locate fixes in specific features is proposed. A future task is to build on this work and extend it towards an approach for fixing delta models over multiple perspectives.

SPLs are commonly represented with the help of feature models depicting commonalities and variability of the system. **Lopez-Herrejon et al.** [LHE10] reuse this knowledge to provide consistency for lower level UML models. A similar approach is presented by **Demuth et al.** [DLHE11]. In combination with the feature models developed in the first contribution of this thesis, an application of these approaches to further prevent inconsistencies during the variant generation would be possible.

4.5 Chapter Summary and Future Work

We have presented a multi-perspective modeling approach for automation systems in accordance to MDD principles. Developers can decide to use textual or graphical modeling editors. Variability is handled through the application of delta modeling to each of the three modeling perspectives. It is possible to generate a complete system variant by applying the corresponding deltas to the core model. The approach is equipped with consistency rules and different methods to enforce them. Overall, we defined two different categories with intra- and inter-perspective for our modeling

perspectives and integrated all of them in our tool chain. Based on these rules, we presented different strategies for consistency checking. Delta modeling itself provided us with the foundation to optimize the product-based consistency checking approach. In the end, we can ensure a valid system variant without any inconsistencies. All presented concepts are applied to our running example of the PPU. By including a code generation for the automation control software of CODESYS, we were also able to successfully evaluate the complete tool chain, from the design phase to the actual execution, on the physical machine.

Future Work. Several tasks are left for future work in this thesis and will be discussed in the following. First of all, to further assess the usability, we need to carry out a controlled user experiment with respect to understandability, learnability, maintainability and general feasibility of the modeling approach. Additionally, we could evaluate the improvement in usability of the graphical modeling approach over the textual one. The scalability can only be evaluated by using a larger case study than the PPU.

The current realization provides developers only with the knowledge that inconsistencies exist in the models. A concept to repair these faults is missing. The idea is to support developers in a way that is similar to common IDE features for programming languages, e.g., adding the import statement for a used function as available in Eclipse or Microsoft Visual Studio. In terms of our modeling approach, a component in the architecture has to be connected to a state machine in the behavior perspective. If there is no such behavior, a repair operation would be to create a default state machine, if the developer complies with the suggestion. The possible cases for inconsistencies have to be explored and fixing operations for these cases have to be provided. Formulating those fixes in terms of deltas allows us to nicely integrate them in the overall modeling framework.

Our modeling approach supports variability in space. It most likely does support variability in time, i.e., evolution, as well, since delta modeling is expressive enough [Sch10]. However, a detailed investigation of evolution is on open point. In addition, we can encounter runtime adaptation introducing a third dimension [GKP+14, DPS12]. The notion of delta modeling can be extended to be applicable dynamically during runtime as previous work has shown for programming languages [DS11, DPS12]. Furthermore, a reference architecture for model-based runtime adaptation that includes a generic framework for models@runtime is available [AGJ+14]. A combination of both aspects within our modeling approach gives us the means to also capture this third dimension of change.

5 Model-Based Performance Analysis for Software Product Lines

This chapter shares material with work published in [KST14], [KTTS15] and [KTTS17].

Contribution

In order to reason about performance properties of an automation system, we extend the UML activity diagrams of the workflow perspective with performance information. Given the annotated models, we devise an efficient family-product-based performance analysis for SPLs and show its superiority compared to a product-based approach. The developed concept is applied to different classes of continuous phase-time distributions with exponential- and Coxian-distributed service times. A general application is presented with two real-world software systems, while a large-scale evaluation is conducted with artificially generated performance models.

Performance is a key requirement in automation systems and user-centric software applications because it directly affects the perceived quality of a system. Software models such as UML diagrams (cf. Chapter 4) allow us to reason about architectural and implementation issues, but non-functional properties of a system are neglected. A performance model can single out the most relevant characteristics of a system that lead to performance degradations causing, e.g., throughput bottlenecks or excessively large response times. Traditionally, the use of performance models is advocated as early as possible in the development lifecycle to drive design decisions towards performance efficiency, rather than fixing performance issues only after deployment, at a higher cost [CDI11]. This typically involves the evaluation of several instances of a model. For example, *what-if* scenarios investigate the impact that changes of certain parameters (service rates or routing probabilities) have on the system's performance. These evaluations have to be efficient, since a significant delay in the development process is not tolerable during design time.

However, the efficiency is threatened by two major sources of complexity which already affect a performance model for a single variant and get increasingly more difficult to handle for a complete product line with thousands of variants.

- *Model-specific complexity:* Many systems typically consist of a large number of components interacting with each other to implement a certain func-

tionality: users, robots, conveyor lines, CPU cores and so on. Traditionally, models for these systems have been developed using techniques based on a discrete state-space representation such as continuous-time Markov chains (CTMCs) [BDIS04]. Unfortunately, these suffer the infamous problem of *state-space explosion*: the number of states grows at worst exponentially with the number of components in the system.

- *Family-related complexity:* The computations performed for the analysis of a specific model instance cannot be re-used to evaluate another model drawn from the same parameter space (e.g., when a service rate value or routing probability is changed). The situation becomes even more problematic when alternative instances have to be examined in case of structural modifications such as the addition or removal of model elements (cf. Chapter 4).

We consider both problems of complexity within our proposed performance analysis and provide an adequate solution. However, we do not consider possible deadlocks in an automation system, e.g., introduced by a collision of workpieces and a (temporary) degradation in performance as a result.

In this chapter, we propose an efficient performance analysis for SPLs to compute non-functional properties such as throughput, utilization, (average) queue length and (average) response time of a system. The analysis is based on the workflow models proposed in Chapter 4 and capable to estimate the performance not only of an automation system, but of any workflow-type software system.

Again, we begin with the necessary background for this contribution consisting of Markov processes, queueing networks and their connection to performance engineering. This is followed by foundations for a naive performance analysis in queueing networks with exponentially and non-exponentially distributed service times. Afterwards, we propose a more efficient performance analysis for both queueing network classes. Next, the developed analysis is evaluated for both types and compared to the naive approach. After a description of the related work, we conclude the Chapter and suggest possible future work in this direction.

5.1 Preliminaries

While UML is one of the standard languages for modeling software systems in a uniform way (cf. Chapter 4 and [MG15]), a similar solution cannot be found for performance modeling. The software performance community introduced several languages and notations, e.g., execution graphs, stochastic Petri nets and process algebras, over the decades that are currently in a co-existing state [CDI11]. For the purpose of this thesis, two of the basic notations with Markov processes and queueing networks are sufficient and will be introduced in the following.

5.1.1 Markov Processes

A Markov process is a stochastic process fulfilling the Markov property. The Markov property states that the probability distribution of future states in the process only depends on the current state and not on past ones [Ste09]. Thus, states have no memory. Markov processes take a central role in the performance analysis of software systems. It is possible to use them as primary notation and directly express the system dynamics with such a Markov process in form of a labeled graph or a transition matrix. However, in many cases Markov processes are used as underlying semantics to find an analytical solution, e.g., in case of queueing networks [BGdMT05]. The latter option also applies to our performance analysis technique, since it relies on the transformation of a software model into a queueing network as our considered performance model. In general, a stochastic process is defined as follows:

Definition 5.1: Stochastic Process

A stochastic process is a collection of random variables

$$X = \{X(t) : t \in T\}$$

where $X(t) : T \times \Omega \to S$. Ω is the probability space, T is usually referred to as time and S is the state space [CDI11, Ste09].

Furthermore, stochastic processes can be categorized using the state space (discrete or continuous) and the time (also discrete or continuous). The state space S of a process is the set of all values that the random variables $X(t)$ can possibly assume. Hence, each individual value represents one state in the process.

Definition 5.2: Markov Process

A stochastic process is called a Markov process if $\{X(t)\}$ is memoryless, which can be expressed in a mathematical way by

$$\begin{aligned} Pr(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n, \ldots, X(t_1) = x_1) \\ = Pr(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n) \end{aligned}$$

In a more simplified way, we can express this property as the probability to go from the present state $s(t_n)$ to a future state $s(t_{n+1})$ which is always identical regardless if we consider the complete history of the process or just the present state [CDI11].

The graph in Fig. 5.1 represents a Markov process with two states. We can switch between states based on the specified probability. It is also possible to express this

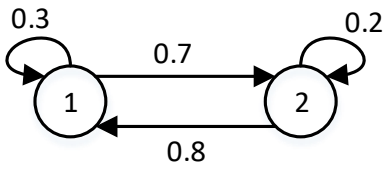


Figure 5.1: Markov process transition graph.

system in matrix notation with:

$$P = \begin{bmatrix} 0.3 & 0.7 \\ 0.8 & 0.2 \end{bmatrix}$$

Each row i and each column j represent a state and $P(i, j)$ gives us the probability to transition from one state to another. In any case, the outgoing probabilities of a state must always sum up to 1.

The primary goal of analyzing such a Markov process is the calculation of a probability distribution for the random variable $X(t)$ in the state space S. Ultimately, the system enters a regular pattern of behavior which is also called the steady state of the system. Several performance metrics that we consider throughout this chapter can be derived from this steady state. Markov processes have been extensively researched for many decades and today they are widely used in the field of performance analysis [Ste09, CDI11]. Finding the solution for a Markov process and therefore its steady state is closely related to the matrix representation.

The transition between different states is only one aspect of the system behavior. We have to consider the time spent in each state as well. In literature, they are defined as the *sojourn times* and describe the task of how long it takes a system to process something. By entering a state $i \in S$ at a time t and the transition to a new state takes place at time $t + T$, the sojourn time is represented by T. Sojourn times also fulfill the Markov property and are therefore memoryless. For a continuous-time Markov chain (CTMC) as in Definition 5.2 the probability of a state transition in a time interval is defined as follows:

Definition 5.3: Transition Rate

Assume that a transition from the current state i to the new state j occurs with a probability p_{ij}. The probability p_{ij} can be given by a Markov process transition graph (cf. Fig. 5.1). Due to the Markov property, this rate must only depend on the states i and j. Hence, the transition rate is defined as

$$Pr(X(t) = i \mid X(t + dt) = j) = q_{ij}dt + o(dt), \forall i \neq j, i, j \in S$$

where q_i is the sojourn time specified by the probability distribution function and $q_{ij} = q_i p_{ij}$ [Ste09, CDI11].

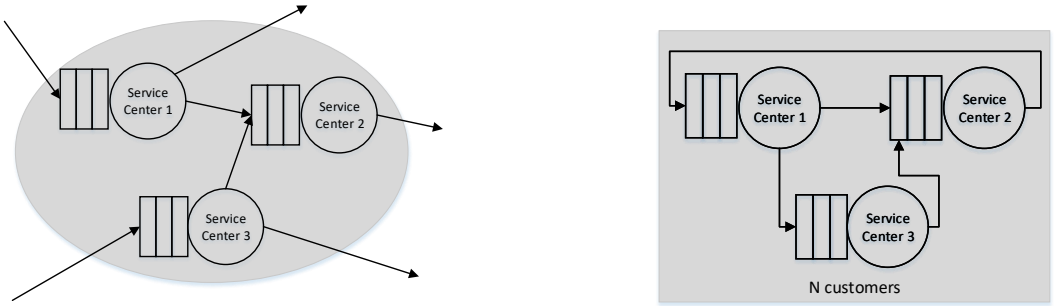
In detail, we will develop an efficient performance analysis based on CTMCs with two different types of probability distributions in the Sections 5.2.1 and 5.2.2.

5.1.2 Queueing Networks

Queueing network (QN) models are a common representation for a performance model. The analysis of a QN combines an efficient model evaluation with an adequate accuracy of the result, making them popular in the software performance evaluation community [Ste09, CDI11]. Informally, a QN is a set of service centers that interact with each other. A service center represents a resource of the system. Customers traverse these service centers and occupy their resources for a specific amount of time. In case of the PPU, we can also identify such service centers, e.g., the crane and stack components. The different workpieces represent customers that are processed by the PPU. Typically, QN are visualized as a directed graph whose nodes are service centers and edges depict the potential paths that customers can take in the system.

The literature differentiates between two classes of QNs based on the workload type present in the network. An open QN can harbor any number of customers that arrive from external sources outside the network. Additionally, customers may leave the QN as well. Fig. 5.2a depicts an open QN with three service centers. Each service center has a queue attached that may be infinite. Of course, in real-world systems as in the automation domain, the queue cannot be infinite. For instance, the stack of the PPU can have a maximum queue of five workpieces while the crane has just a queue of one workpiece. The workload of a closed QN is completely determined by the fixed number of customers circulating the system. It is not possible for customers to leave the QN and no new customers can arrive from external sources. However, a network in which a leaving customers is instantly replaced by a new one, is also considered to be a closed QN. Fig. 5.2b shows an exemplary closed QN with three service centers. Our contribution addresses both QN types with open QN in Section 5.2.1 and closed QN in Section 5.2.2. Queues always follow a scheduling strategy which is first-come-first-serve in all cases throughout this thesis.

The last important parameter in a QN is the service time given by a probability distribution function. It can be informally described as how long it takes a service center to process a customer (cf. sojourn times). We applied our efficient performance analysis to two types of phase-type distributions. These service time distributions are based on a mixture of multiple phases and each phase contains an exponentially distributed service time. In the first case (cf. Section 5.2.1), we consider just one phase represented by a single exponential distribution. The second more complex case is represented by a Coxian distribution that can have 2 or more phases (cf. Section 5.2.2). It is not mandatory that the individual phases are identical. Dis-



(a) Open Queueing Network.

(b) Closed Queueing Network.

Figure 5.2: Classes of Queueing Networks.

tributions with multiple phases are extremely important in the performance evaluation domain, since they are able to approximate any type of probability distribution and thus enable engineers to model more realistic scenarios [BGdMT05, Ste09]. While such a distribution is over the top for a small automation system as the PPU, one can imagine Internet network traffic between several servers as an example. Typically, there are peak times, e.g., in the evening, where many people put load onto the system, and there are less frequented times. We can approximate the real-world traffic by using multiple of such phases as in the Coxian distribution and compute performance metrics in the system [BHLM13, GF99, Ste09]. All exponential distributions give rise to CTMCs, since a discrete time cannot sufficiently express such behavior. However, we consider only a discrete state space, since our main interest lies in modeling and analyzing real-world automation or software systems with distinctive physical components. In addition, we limit ourselves to a single class of customers.

After modeling a QN with all the necessary parameters from service stations to probabilities, we can compute the performance metrics in the steady state behavior of the system. Further mathematical details are omitted at this point, since they will be introduced step-wise throughout the next sections.

We decided to use QN as performance models, since an immediate mapping with software models from the UML is possible [CDI11, ITT16]. This property is especially beneficial for us, since we already have developed UML models that are enriched with a variability modeling mechanism in the previous Chapter 4.

5.2 Foundations

In the following, we describe the two considered queueing network classes in more detail. Additionally, we extend our workflow models with performance annotations to enable reasoning about performance metrics. Given the annotated workflow models, we also present how a naive performance analysis for individual variants is possible in both network classes.

5.2.1 Queueing Networks with Exponential Service Times

The development of an efficient performance analysis for product lines begins with an adequate selection of a QN class as performance model. Our choice is the class of Jackson networks as they are particularly simple to compute due to an efficient product-form solution [Jac63]. A product-form solution means that the computation of a specific metric in this QN, e.g., the throughput, is computationally inexpensive and often possible in polynomial time [Ste09, CDI11]. This property is invaluable with regard to product lines and their possibly massive variant space. A Jackson network is an open QN in which the service times follow an exponential distribution.

Given our multi-perspective modeling approach proposed in the previous chapter, a performance analysis would be possible on all three perspectives [BMB+15, CM02, CDI11]. However, our focus is to provide developers as early as possible with feedback about optimization potential in the system, which is why we use the workflow perspective comprised of UML activity diagrams as source for our performance analysis. The PPU serves as a first application example, and, for convenience, we show once again the **Basic** variant as UML activity diagram in Fig. 5.3. Since the UML is not natively equipped with features to capture performance-related attributes, we have to extend our workflow modeling perspective, respectively.

Figure 5.3: **Basic** variant of the PPU.

Augmentation of the Workflow. A formal definition of the workflow perspective is already available in Section 4.2.2. We extend the previous definition in order to incorporate performance annotations as well. The augmented software models are called performance-annotated activity diagrams. The annotations are independent from the actual annotation mechanism that may be used in an implementation, e.g., as part of our previously developed DSLs or with the MARTE profile (see [Obj11]) and its PaStep stereotype (e.g., [TG08]).

Definition 5.4: Performance-Annotated Activity Diagram for Jackson QN

Let $\mathcal{V}$ be the set of all nodes. A performance-annotated activity diagram for Jackson QNs ($PAAD^J$) is a tuple

$$PAAD^J = (V, E, \lambda, \mu),$$

where $V \subseteq \mathcal{V}$, $E \subseteq V \times \mathbb{R}_{\geq 0} \times V$, $\lambda : V \to \mathbb{R}_{\geq 0}$, and $\mu : V \to \mathbb{R}_{>0}$.

Again, this definition specifies a directed graph annotated with three distinct pieces of information. Each edge $e \in E$ has a non-negative real, giving the probability with which that path is taken by a job in the source node. Each node $v \in V$ is associated with a *service rate*, $\mu(v)$, denoting the average speed at which a job is processed in v. Finally, $\lambda(v)$ denotes the *workload*, the speed at which jobs arrive from the external world due to the open QN class. Using this definition, we can change the software model of the PPU in Fig. 5.3 into a performance model based on a Jackson network as depicted in Fig. 5.4. In case of the PPU, all performance annotations are based on actual measurements conducted at the physical machine. For each node v, the top-left label gives $\lambda(v)$ and the top-right label represents $\mu(v)$. The edges are labeled with their associated probabilities. Workpieces can only arrive at the stack component of the PPU, which is why all other nodes have an arrival rate of 0 due to the open QN nature. Of all three components, we can easily identify the crane as the component with the slowest service time.

We point out that Definition 5.4 does not explicitly consider initial, final, choice, and merge nodes. Similar to previous work [TG08], we argue that these are not necessary when an activity diagram is used as a performance model. For instance, a node with no outgoing edges can be interpreted as a node where workpieces leave the system after they are processed; this is equivalent to connecting the node to a final node with a probability one. An initial node represents the point in the workflow where the whole process starts; this is equivalently represented by labeling nodes through the function λ, which models how workpieces appear at each node. Moreover, if the probabilities attached to the outgoing edges of a merge node sum up to a value $p < 1$, this can be interpreted as a workpiece that leaves the system with probability $1-p$; this is equivalent to adding a further edge to the merge node, labeled with $1 - p$, which leads to a final node. In a similar fashion, decision nodes are not necessary as they can be modeled using multiple transitions from the same node. Thus, we simply remove the nodes that are not supported and redirect the edges appropriately in our $PAAD^J$s.

Finally, of all the elements in UML activity diagrams that are not used in Definition 5.4, we wish to remark the absence of fork and join nodes. Unfortunately, their performance interpretation leads to models that, in general, do not have an efficient solution for which we strive. Thus, we neglect them throughout this contribution and leave further investigation of this matter to future work.

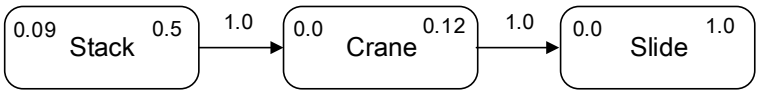


Figure 5.4: $PAAD^J$ of the **Basic** variant.

Next, we provide necessary conditions that have to be fulfilled by a $PAAD^J$ in order to yield a meaningful performance model.

Definition 5.5: Well-formedness

A $PAAD^J$ is well-formed if and only if the following conditions hold:

i) There exists at least one $v \in V$ such that $\lambda(v) > 0$;

ii) For all $v \in V$ it holds that $\sum_{(v,p,v') \in E} p \leq 1$;

iii) For all $v, v' \in V$, for any $(v, p, v'), (v, q, v') \in E$ it holds that $p = q$;

iv) There exists at least one $v \in V$ such that $\sum_{(v,p,v') \in E} p < 1$.

Assumption i) is required to ensure that the model receives customers starting at least from one node. Assumption ii) corresponds to the natural interpretation of edge labels as probabilities. Assumption iii) requires that there is at most one directed edge between any two nodes, so that the probability with which node v' is visited after v is not ambiguously defined. Finally, iv) requires that, eventually, jobs leave the workflow. This is a necessary condition for a steady-state behavior. Otherwise, the system would keep accumulating customers or workpieces [KST14].

Product-Based Performance Analysis. The main objective of a performance analysis is to compute metrics such as throughput, utilization, queue lengths or response times. A necessary requirement is the construction and solution of the famous traffic or flow balance equations. Traffic equations describe the mean arrival rate of customers at a node and their solution provides us with the means to compute the other previously mentioned performance metrics. A Jackson network is characterized by the number of external and internal arrivals at each node and literature already gives us the essential traffic equations for a product-based performance analysis [Ste09, Jac63]. Thus, we can analyze each variant in isolation given a well-formed $PAAD^J$ with:

Definition 5.6: Product-based Evaluation

The product-based evaluation of a $PAAD^J$ is given by the following traffic equations

$$(I - P^T)\gamma = \lambda, \tag{5.1}$$

where I is the identity matrix, λ is the vector containing all arrival rates at each node $\lambda(v)$. This is ordered in the same way as nodes appear in the matrix

P, which is defined as $P = (p_{v,v'})$, for all $v, v' \in V$ with

$$p_{v,v'} = \begin{cases} p & \text{if } (v, p, v') \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Finally, γ is the vector of unknown arrival rates, with elements denoted by $\gamma(v)$ [KST14].

In essence, we are interpreting a $PAAD^J$ as a CTMC that underlies a Jackson network [Jac63], by giving the following semantics [KST14]:

- $\lambda(v)$ is the *arrival rate* of customers at the node v, which is exponentially distributed. If $\lambda(v) = 0$, node v does not have exogenous arrivals and may only process customers arriving from other nodes, according to the topology of the workflow.
- Customers at a node v are processed by a *service rate* $\mu(v) > 0$, also according to an exponential distribution. When the node is busy serving a customer, the other customers accumulate in a queue and are scheduled according to a first-come-first-serve strategy.
- P is the *routing probability matrix*, defining with which probability a job in node v, after being serviced, moves to any other node v'.
- γ gives the *effective* arrival rates, which take into account the actual traffic incoming at node v due to the exogenous arrival as well as to the feedback from other nodes.

Recalling the $PAAD^J$ of the PPU in Fig. 5.4 we get the following elements:

$$P = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \qquad \lambda = \begin{bmatrix} 0.09 \\ 0.0 \\ 0.0 \end{bmatrix} \qquad \mu = \begin{bmatrix} 0.5 \\ 0.12 \\ 1.0 \end{bmatrix} \tag{5.2}$$

The mean service time is given by $\mathbb{E} = 1/\mu(v)$. It is similar for the mean arrival time $\tau = 1/\lambda(v)$. Assuming the time unit to be in seconds, a service rate of 0.5 means that the stack components needs 2 seconds on average to process a workpiece, while the crane components takes about 8.3 seconds on average to completely process a workpiece. The mean arrival rate between two workpieces at the stack is calculated by $\tau_{St} = 1/0.09 = 11.1\ s$. Already at this point, we can derive the information that the stack will inevitably run out of workpieces, since the arrival rate is slower than the processing rate of the slowest component, namely the crane. However, this

observation is only possible due to the simple sequential order of processing steps in the **Basic** variant of the PPU.

Returning to the traffic equations, once the system (5.1) is solved for γ, the steady-state behavior of the network is completely characterized (e.g., [Ste09]). However, these traffic equations are only valid and can be solved if the QN is stable [Mit97]. The transient behavior, i.e. the time before the system settles into the steady state, is not considered in this thesis. Specifically, the following metrics can be computed for any $v \in V$.

- $\gamma(v)$ is the *throughput*, i.e., the rate at which customers are served at node v. In the steady state, the throughput is equal to the effective arrival rate, since the rate of incoming and outgoing customers is balanced at a node.
- The *utilization* of node v, denoted by $\rho(v)$, i.e., the probability that the node is busy serving customers, is given by:

  $$\rho(v) = \gamma(v)/\mu(v).$$

  This also induces the stability requirement with $p(v) < 1$, since we have only one server at each node available to process customers. If this condition does not hold, the QN would accumulate customers and per definition we are no longer in the steady state.
- The *queue length* at node v, denoted by $L(v)$, i.e., the number of customers at node v including those in service, is given by

  $$L(v) = \rho(v)/(1 - \rho(v)). \tag{5.3}$$

- The *average response time* of customers at node v, denoted by $W(v)$, is given by

  $$W(v) = L(v)/\gamma(v).$$

For instance the utilization of the PPU with the given values in the steady-state is

$$\begin{aligned}\rho(Stack) &= 0.09/0.50 = 0.18 = 18\% \\ \rho(Crane) &= 0.09/0.12 = 0.75 = 75\% \\ \rho(Slide) &= 0.09/1.00 = 0.09 = 9\%\end{aligned}$$

which clearly identifies the crane component as the bottleneck of the PPU.

5.2.2 Queueing Networks with Coxian Service Times

In this section, we extend the concepts previously applied to Jackson networks, i.e., enhancing the workflow with performance annotations and a product-based performance analysis, to a more powerful class of QN. In particular, we consider a QN where service times are defined by Coxian distributions. These can be informally considered as a "composition" of exponential *phases* that can approximate any given probability distribution arbitrarily closely while still keeping the whole network representable as a CTMC [Cum82, Ste09]. Such general distributions are necessary to model realistic data traffic in networks, e.g., Internet traffic between data centers. A Jackson network has only one exponential phase and cannot deal with such scenarios. In addition, we lift the restriction of parallelism/concurrency by modeling multiple, independent and identical servers at each service station, e.g., to model multi-core CPUs. However, both extensions make an efficient solution impossible as in Jackson networks, since we encounter the problem of state-space explosion. The number of states grows rapidly, exponentially in the worst case, with the number of customers, service centers, servers and phases in the CTMC. A modeler has to resort to expensive simulations in these cases, which is inefficient considering a product line with a large variant space. We tackle this problem in a closed QN by providing an approximation to the traffic equations in order to achieve an efficient solution. As in case for Jackson networks, the size should then only depend on the network topology.

In this case, the PPU is not sufficient to serve as running example. There are no multiple servers at a component. The processing rate of workpieces is always identical for an individual variant making phase-type distributions not necessary. This is why we introduce a new running example in the following and apply the all concepts to this example.

Adaptation of the Workflow Augmentations. In order to successfully model a QN with Coxian values, we have to adapt the workflow modeling perspective again. A $PAAD^J$ representing a Jackson network does not include multiple servers at service stations and several phases in the service time distribution, which are necessary for the new QN class. Thus, we introduce an adapted performance-annotated activity diagram, referred to as $PAAD^C$, which can represent Coxian values as well as multiple servers. However, the main restriction of $PAAD^J$s remains with the lacking support for fork/join nodes [DMI04, BM05, BMB+15]. Thus, we have no parallelism for the path which a customer can take in the system, but service stations contain parallelism under the aspect of multiple servers.

We now present the formal definition of the adapted $PAAD^C$ analogue to Definition 5.4:

Definition 5.7: $PAAD^C$ with Coxian Values

A $PAAD^C$ is a tuple $PAAD^C = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{S}, \mu, p)$ where:

$\mathcal{V} \subseteq \mathbb{N}$ is the set of nodes; in the following we shall use $\mathcal{V} = \{1, 2, \ldots, n\}$;

$\mathcal{E} \subseteq \mathcal{V} \times [0; 1] \times \mathcal{V}$ is the set of labeled edges;

$\mathcal{S} = (S^1, \ldots, S^n) \in \mathbb{N}^n_{>0}$ is the server multiplicity vector;

$\mathcal{C} = (C^1, \ldots, C^n) \in \mathbb{N}^n_{\geq 0}$ is the initial condition vector;

$\mu = (\mu^1, \ldots, \mu^n)$, with $\mu^i = (\mu^i_1, \ldots, \mu^i_{m_i}) \in \mathbb{R}^{m_i}_{>0}$ for $m_i > 0$ are the Coxian rate vectors;

$p = (p^1, \ldots, p^n)$, where $p^i = (p^i_1, \ldots, p^i_{m_i}) \in \mathbb{R}^{m_i}_{\geq 0}$ with $p^i_{m_i} = 1$ for $1 \leq i \leq n$ are the Coxian probability vectors.

We remark that m_i is the number of phases of the Coxian distribution for node i; the Coxian vectors for station i are denoted by $\mu^i = (\mu^i_1, \ldots, \mu^i_{m_i})$ and $p^i = (p^i_1, \ldots, p^i_{m_i})$. With this notation, the mean service time at station i is (cf. [Ste09]):

$$\mathbb{E}^i = 1/\mu^i_1 + p^i_1/\mu^i_2 + p^i_1 p^i_2/\mu^i_3 + \ldots + p^i_1 \cdot \ldots \cdot p^i_{m_i-1}/\mu^i_{m_i}. \tag{5.4}$$

Information Retrieval System. Next, we introduce our new running example of the *Information Retrieval System* (IRS) taken from [CM02]. The application scenario of an IRS is as follows: Users can access the IRS via a main interface or terminal stations and perform search requests. The IRS itself can execute two types of operations consisting of local and remote data research. If the user requires a local research, the IRS accesses a local database or disk searching for the element of interest. In case of a remote research, the item is searched over the network. As we may have multiple users accessing the IRS and executing remote researches, all queries are evenly divided onto three processors to ensure a good response time of the IRS. We chose this model, because it is a realistic example that already is available in the literature and we can directly model it as a $PAAD^C$. **Cortellessa et al.** [CM02] also modeled a second variant of the IRS based on structural modifications, which we can capture with the delta operations in our framework. Figure 5.5 shows a graphical representation of the $PAAD^C$ for the *centralized platform* variant of the IRS of [CM02, Fig. 15]. We take this as our core variant. All hardware and delay entities can be directly represented as nodes in our $PAAD^C$. Hardware enti-

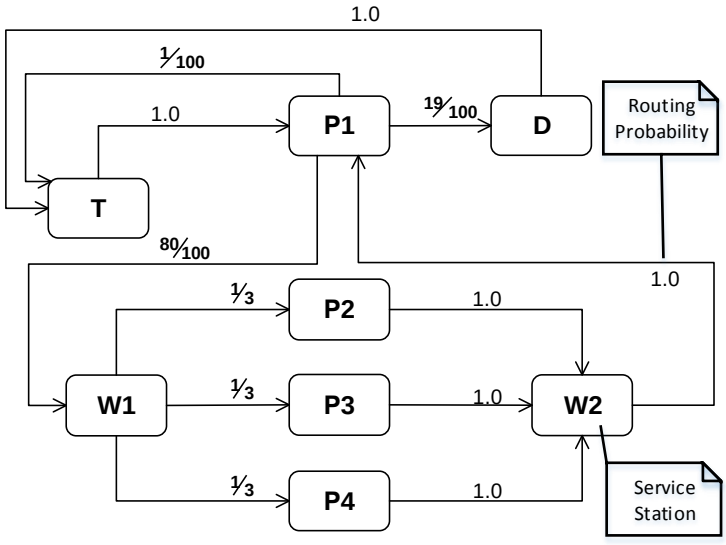


Figure 5.5: A Performance Annotated Activity Diagram for the IRS *centralized platform* case study of [CM02]. We refer to Table 5.1 for the remaining performance annotations.

ties represent processors and disks, while delay entities refer to terminals and the network.

Each node, labeled with a boldface symbol, represents a service station and is annotated with the following information: number of customers at that station in the initial condition, server multiplicity, and service time distribution. Edges are annotated with probabilities, with the usual interpretation: for instance, a customer served at the CPU station **P1** will go to the disk station **D** with probability 19/100, or into the wide-area network station **W1** with probability 80/100; else it goes back to terminal station **T** [KTTS17].

Table 5.1 shows the performance annotations for all service stations of the running example in Fig. 5.5. We reused the mean service times for individual entities given

Table 5.1: Performance annotations for the core $PAAD^C$ of Fig. 5.5.

Name	*Customers*	*Servers*	*Rates*	*Probabilities*
T	10	10	(1,2)	(1/2,1)
D	0	1	(4000, 2000)	(1/2,1)
P1	0	4	(20000, 10000)	(1/2,1)
P2	0	4	(20000, 10000)	(1/2,1)
P3	0	4	(20000, 10000)	(1/2,1)
P4	0	4	(20000, 10000)	(1/2,1)
W1	0	10	(1,2)	(1/2,1)
W2	0	10	(1,2)	(1/2,1)

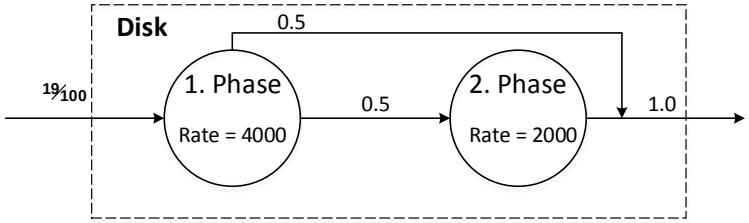


Figure 5.6: Coxian CTMC for the **Disk** service station.

in [CM02], e.g., for the stations **D** and **P1**. The only nonstandard annotation concerns the service time distribution, represented by the vectors in the last two columns of the table. Such vectors, hereafter called *Coxian rate vector* and *Coxian probability vector*, respectively, provide a representation of a CTMC that describes the service time at a station. The length of the vectors gives the number of states of such CTMC (the *phases* of the distribution). For instance, we always consider two phases in our running example. The Coxian rate vector lists the rate of the exponentially distributed residence time at each state; the Coxian probability vector gives the probability with which the service process moves from one state to the next. If such probability, p, is less than one, the service ends after the current state with probability $1 - p$. The time between the start of the process in the first state and its exit from any state determines the non-exponential distribution for the service. For instance, Fig. 5.6 shows a CTMC representation for a Coxian distribution with two phases of the **Disk** service station. Starting from phase 1, a customer has a residence time of 4000, but with probability 1/2 it will be followed by an additional exponential delay with rate 2000. Otherwise the second phase is skipped and a customer leaves the service station after the first phase.

Finally, the mean service times in seconds for the stations **D** and **P1** are calculated by (cf. [CM02], the values are identical):

$$\mathbb{E}^D = \frac{1}{4000} + \frac{1/2}{2000} = 0.0005 \ s$$
$$\mathbb{E}^{P1} = \frac{1}{20000} + \frac{1/2}{10000} = 0.0001 \ s$$

We continue with the mandatory conditions to achieve a meaningful $PAAD^C$ model:

Definition 5.8: Well-formedness

A $PAAD^C$ is well-formed if and only if the following conditions hold:

i) $\sum_{(i,r,j')\in\mathcal{E}} r = 1$ for all $i \in \mathcal{V}$;

ii) Any pair $(i, r, j), (i, r', j) \in \mathcal{E}$ yields $r = r'$. Hence, we let $r_{i,j}$ be the unique probability such that $(i, r_{i,j}, j) \in \mathcal{E}$.

iii) Any two nodes of a $PAAD^C$ are connected by a path with non-zero probability. Put different, the nodes of a $PAAD^C$ build a strongly connected graph.

Condition *i)* imposes a closed topology: with probability 1 a customer serviced at any station goes into some other station; condition *ii)* requires that there exists only one edge between two nodes. Considering our *centralized platform* IRS running example, we modeled a well-formed $PAAD^C$. *iii)* means that the routing probability matrix is irreducible. In comparison to the well-formedness of a $PAAD^J$, it is no longer possible for customers to leave the QN and customers can always reach any station in the QN regardless of their current position.

From a CTMC to Traffic Equations. At this point, we were able to perform a product-based performance analysis for Jackson networks given the traffic equations [Jac63]. However, a performance analysis for a QN with Coxian values is only possible with an expensive simulation of the complete system due to the state-space explosion in the CTMC. A compact and efficient representation such as the traffic equations is not available [BGdMT05]. Fortunately, we can perform an approximation of the traffic equations based on ordinary differential equations. The general procedure to acquire the desired traffic equations is depicted in Fig. 5.7.

We first interpret a $PAAD^C$ as a closed QN with multi-server queues and Coxian-distributed service times. A node i has a queue with S^i servers. If the number of customers in station i at time t, denoted by $C^i(t)$, is less or equal S^i, each customer is serviced with a Coxian-distributed service time that is specified by vectors μ_i and p_i. Instead, if $C^i(t) > S^i$ then the number of customers that are queueing for service at time t is at least $C^i(t) - S^i$. All stations have a first-come-first-serve strategy. Finally, $R = (r_{i,j})_{1 \leq i,j \leq n}$ is the routing probability matrix, defining with which probability a customer in node i, after being serviced, moves to any node j.

Since the service-time distribution at each station consists of multiple exponential stages, the whole QN can be described as a CTMC. Intuitively, the state of the system is completely characterized by how many customers are present at each station (the queue length); in particular, the state has to keep track of how many customers are executing which phase of service. A CTMC state is characterized by a so-called *population vector* (cf. Fig. 5.7). For instance, the initial state of the CTMC in Fig. 5.5 has 0 customers in all stations but **T**, where all 10 customers are serviced in phase 1 of the two-phase Coxian distribution; indeed, since the

Figure 5.7: Workflow from a CTMC to the traffic equations.

topology is closed and there are 10 circulating customers, stations **T**, **W1** and **W2** are effectively acting as *delay stations* where customers do not compete for shared resources. We can define such population vectors for each state of a station in the network and their combination gives us a description of the complete network state at the current point in time. The logical next step is to define the transition rates (cf. Definition 5.3) with which a customer goes from one state to another state in the CTMC [KTTS15, KTTS17].

Again, in contrast to a Jackson network, it is not possible to derive the concrete traffic equations based on the CTMC model, i.e., routing probability matrix and transition rates. However, **Kurtz** [Kur70] developed a method to approximate the CTMC by a system of ordinary differential equations (ODE) and proved that it is a sound approach. We are able to use this approximation under the assumption of an irreducible routing probability matrix which in turn is a requirement for a well-formed $PAAD^C$. Here, we limit ourselves to providing the resulting ODEs.

The ODE representation of a $PAAD^C = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{S}, \mu, p)$ is the set of ODEs $\Xi_1 \cup \ldots \cup \Xi_n$, where m_i denotes the number of Coxian phases and Ξ_i is defined by

$$\begin{aligned}
\dot{C}_1^i &= -\mu_1^i \min(C_1^i, S_1^i) + \sum_{j=1}^{n} r_{j,i} \Big[(1-p_1^j)\mu_1^j \min(C_1^j, S_1^j) + \\
&\qquad + \sum_{l=2}^{m_j-1} (1-p_l^j)\mu_l^j S_l^j + \mu_{m_j}^j S_{m_j}^j \Big] \qquad (5.5) \\
\dot{S}_1^i &= -p_1^i \mu_1^i \min(C_1^i, S_1^i) + \sum_{l=2}^{m_i-1} (1-p_l^i)\mu_l^i S_l^i + \mu_{m_i}^i S_{m_i}^i \qquad (5.6) \\
\dot{S}_2^i &= -\mu_2^i S_2^i + p_1^i \mu_1^i \min(C_1^i, S_1^i) \\
\dot{S}_3^i &= -\mu_3^i S_3^i + p_2^i \mu_2^i S_2^i \\
&\vdots \\
\dot{S}_{m_i}^i &= -\mu_{m_i}^i S_{m_i}^i + p_{m_i-1}^i \mu_{m_i-1}^i S_{m_i-1}^i
\end{aligned}$$

and the initial conditions are given by $C_1^i(0) = C^1$, $S_1^i(0) = S^1$, $S_2^i(0) = 0$, …, $S_{m_i}^i(0) = 0$.

Each set of ODEs Ξ_i refers to the equations for station i. This approximation has already been successfully applied in process algebra [TGH12, TDGH12, Hil96] and layered queueing networks [FAOW+09]. Thus, we can describe each station with a set of ODEs [KTTS15, KTTS17].

For instance, we now present a fragment of the ODEs for the IRS. We denote by C_1^{P2} the number of customers (C) in station **P2** (superscript) that are in stage 1 (subscript) of the Coxian distribution; and similarly, by S_1^{W1} the number of servers that are serving the customers in the first stage of the Coxian distribution in sta-

tion **W1**. The ODEs for these two variables evolve according to:

$$\dot{C}_1^{P2} = -2 \cdot 10^4 \min(C_1^{P2}, S_1^{P2}) + \frac{1}{6}\min(C_1^{W1}, S_1^{W1}) + \frac{2}{3}S_2^{W1} \tag{5.7}$$

$$\dot{S}_1^{W1} = -\frac{1}{2}\min(C_1^{W1}, S_1^{W1}) + 2S_2^{W1} \tag{5.8}$$

The complete ODE semantics are available in the literature [KTTS15, KTTS17].

Product-Based Performance Analysis. The ODE representation lets us derive the traffic equations for the steady-state throughput of a $PAAD^C$. We can achieve this by setting each ODE to zero and plugging (5.6) into (5.5) (cf. [KTTS15] and [KTTS17] for more details here). Finally, the traffic equations are given in matrix notation with

$$R^\dagger \zeta = \zeta, \qquad \zeta = (\zeta^1, \ldots, \zeta^n)^\dagger. \tag{5.9}$$

where $(\ \cdot\ )^\dagger$ denotes matrix transposition. R is the routing probability matrix and ζ is the vector of unknown arrival rates.

This system of equations is the ODE counterpart of the traffic equations [BGdMT05], which relate station throughputs through the routing probability matrix. Thus, we can compute performance metrics for individual variants in a product-based manner again. However, we still cannot recover the steady state throughputs T^i from these equations. This is because, we have to deal with an homogeneous system of equations, which has infinitely many solutions. We can compute normalized arrivals by setting, e.g., $\zeta^1 = 1$ (other choices are also possible) and find a normalized solution. These arrivals are typically called *relative arrival rates*, i.e., the values $\zeta^1/\zeta^1, \ldots, \zeta^n/\zeta^1$ [BGdMT05]. For instance, the relative arrivals rates for the core IRS variant are given by

$$(\zeta^T, \zeta^{P1}, \zeta^D, \zeta^{W1}, \zeta^{W2}, \zeta^{P2}, \zeta^{P3}, \zeta^{P4}) = (1.0, 5.0, 0.95, 4.0, 4.0, 1.3, 1.3, 1.3)$$

However, by exploiting the fact of a closed QN, we can define a bottleneck condition similar to the stability condition for Jackson networks (cf. Section 5.2.1) and derive the actual throughputs of each station. After the following bottleneck condition is checked for all service stations:

$$S^i > \frac{\mathbb{E}^i \zeta^i}{\sum_{j=1}^n \mathbb{E}^j \zeta^j} \sum_{j=1}^n C^j, \quad \text{for all} \quad 1 \leq i \leq \nu \tag{5.10}$$

We can derive the steady-state throughput as follows:

Definition 5.9: Throughput

For all stations i, with $1 \leq i \leq n$, the steady-state throughput T^i is

$$T^i = \begin{cases} \zeta^i (\sum_{j=1}^{n} \mathbb{E}^j \zeta^j)^{-1} \sum_{j=1}^{n} C^j & if\ (5.10)\ holds, \\ \zeta^i \rho & else,\ where\ \rho := \min \left\{ S^j (\zeta^j \mathbb{E}^j)^{-1} \;\middle|\; 1 \leq j \leq \nu \right\} \end{cases}$$

The condition (5.10) searches for bottlenecks in the $PAAD^C$ by relating the processing rate of the considered service station i to the complete network. In particular, it reflects the worst case scenario, since all customers are present at the considered station. The question is now, if the station i can process the customers faster in comparison to the complete network or if they already queue again in station i. A queue is comprised of all customers in service as well as customers waiting for service. If the queue exceeds the number of available servers at a service station, we encounter a bottleneck and (5.10) is violated. It is possible that multiple stations pose a bottleneck in which case the actual throughput of all stations is limited by the slowest station in the network (second case). If no bottleneck is present in the network the relative arrival rate ζ^i with regard to the processing rate of the complete network gives us the actual throughput (first case). Again, the throughput in the steady-state of a system is balanced (number of incoming customers equals the number of outgoing customers). Thus, we do not have to incorporate the mean service time $\mathbb{E}$ here. A proof of this result is available in [KTTS17].

For instance, in the $PAAD^C$ model of Fig. 5.5 we compute the steady-state throughputs for each station in our running example with

$$(T^T, T^{P1}, T^D, T^{W1}, T^{W2}, T^{P2}, T^{P3}, T^{P4}) = \\ (0.88, 4.44, 0.84, 3.55, 3.55, 1.18, 1.18, 1.18)$$

T^{P1} represents the steady-state throughput in the station **P1** calculated as 4.44 customers per second (on average). Furthermore, we can compute the queue length with the obtained results. Since queue i has S^i servers and the mean service rate is given by $\mathbb{E}^i$, its maximal throughput is $S^i/\mathbb{E}^i$.

Definition 5.10: Queue Length

If (5.10) holds, we have that the steady-state queue length at station i is given by $L^i = \mathbb{E}^i T^i$ for all $1 \leq i \leq n$. If, instead, (5.10) is violated, we have that

$$\sum_{i \in I} L^i = \sum_{j=1}^{n} C^j - \sum_{i \notin I} \mathbb{E}^i T^i \qquad for\ I := \left\{ 1 \leq i \leq \nu \mid S^i (\zeta^i \mathbb{E}^i)^{-1} = \rho \right\}$$

$$L^i = \mathbb{E}^i T^i \qquad for\ any\ i \notin I$$

By noting that $i \in I$ whenever $S^i/\mathbb{E}^i = \zeta^i \rho = T^i$, we conclude that I denotes the bottlenecked queues of the network. We can only quantify the number of customers residing in all bottlenecked queues $\sum_{i \in I} L^i$. For instance, in the $PAAD^C$ model of Fig. 5.5, we compute the steady-state throughputs with

$$(L^T, L^{P1}, L^D, L^{W1}, L^{W2}, L^{P2}, L^{P3}, L^{P4}) = (1.1110, 0.0004, 0.0004, 4.4439, 4.4439, 0.0001, 0.0001, 0.0001)$$

We can observe that the system has no bottleneck, since all queues are below the number of available servers at a service station. In addition, we can see that the delay stations **T**, **W1** and **W2** have the largest queues. This is due to their slow mean service rate of 1.25 compared to the other service stations. Indeed, if we raise the number of customers to 22 the queues of **W1** and **W2** increase to 9.8 almost reaching the server capacities. In case of 23 customers both stations pose a bottleneck. These numbers were determined by continuously raising the customers in the system from the initial value of 10 to 23 in incremental steps of one.

As stated in [KTTS15, KTTS17], this allows to propose the following definition, which permits the analysis of a well-formed $PAAD^C$.

Definition 5.11: Product-based Evaluation

The product-based evaluation of a $PAAD^C$ is given by the expressions for $T^1, L^1, \ldots, T^n, L^n$ as described in Def. 5.9 and Def. 5.10.

Hence, we can perform a product-based performance analysis for Jackson networks as well as networks with a Coxian service time distribution concluding the foundations.

Prof. Mirco Tribastone and Assistant Prof. Max Tschaikowski contributed significantly to the theoretical concepts of this part, especially the ODE approximation representing the traffic equations is their achievement.

5.3 Creating the 150%-Variability Model for Jackson Networks

At this point, we can model a single variant of a software or automation system with the necessary performance annotations and perform a product-based performance analysis. In this section, we propose an efficient performance analysis for the Jackson networks making the analysis of a complete product line feasible.

5.3.1 Application of Delta Modeling

We already know how to model additional variants of the considered system with the help of delta modeling without performance-related values in our workflow (cf.

Chapter 4). Thus, the obvious next step is to extend the concept of delta modeling to also cover performance aspects. Again, each delta contains a set of basic operations to be performed on a $PAAD^J$, such as the addition and removal of a node, or the modification of parameters such as the probabilities of an edge or the service rate in a specific node. Applying a delta to the defined core yields a new $PAAD^J$ *variant*, which has performance characteristics that can be analyzed using the product-based evaluation in Definition 5.6. The **Basic** variant of the PPU will serve once again as our core variant.

First, we start with extending all possible atomic delta operations for a $PAAD^J$.

Definition 5.12: *PAAD*^{J} Deltas

A $PAAD^J$ delta is a set of delta operations $\delta \subseteq Op$, where

$$\begin{aligned} Op = {} & \{\mathsf{add}\ (v_i, \lambda_i, \mu_i) \mid v_i \in V, \lambda_i \geq 0, \mu_i \geq 0\} \\ & \cup \{\mathsf{add}\ (v_i, p_{ij}, v_j) \mid v_i, v_j \in V, p_{ij} > 0\} \cup \{\mathsf{rem}\ v \mid v \in V\} \cup \{\mathsf{rem}\ e \mid e \in E\} \\ & \cup \{\mathsf{mod}\ \lambda(v_i)\ \mathsf{by}\ \lambda_j \mid v_i \in V, \lambda_j \geq 0\} \cup \{\mathsf{mod}\ \mu(v_i)\ \mathsf{by}\ \mu_j \mid v_i \in V, \mu_j > 0\} \\ & \cup \{\mathsf{mod}\ (v_i, p_{ij}, v_j)\ \mathsf{by}\ q_{ij} \mid (v_i, p_{ij}, v_j) \in E, q_{ij} > 0\}. \end{aligned}$$

Adding an action node v also requires to define the arrival rate $\lambda(v)$ and the service rate $\mu(v)$. When an edge is added, its associated probability must be strictly positive. This is without loss of generality because an edge with probability zero essentially corresponds to the case where no edge at all is connecting two nodes. For the same reason, when a service rate or a probability are modified we require strictly positive values. This is not the case for arrival rates, so long as the resulting variant is well-formed in that at least one node has a positive arrival rate. The modification of values is a simplification since it can also be encoded by removing node or edge and adding it with the desired rate or probability, respectively. Analogously to Chapter 4, we only consider a single delta to generate a new $PAAD^J$ variant, and all conditions for applicability as well as consistency must hold [KST14].

The following definition formalizes how to obtain a variant by applying a delta to a $PAAD^J$.

Definition 5.13: *PAAD*^{J} Delta Application

The application of an applicable and consistent delta $\delta \subseteq Op$ to a $PAAD^J = (V, E, \lambda, \mu)$ is defined by the function $PAAD^{J'} = apply(PAAD^J, \delta)$, where $PAAD^{J'} = (V', E', \lambda', \mu')$. It is recursively defined as follows.

1. Case $\delta = \emptyset$: $PAAD^{J'} = PAAD^J$.
2. Case: $\delta = \delta' \cup \delta'' \wedge \delta', \delta'' \in Op$: $PAAD^{J'} = apply(apply(PAAD^J, \delta'), \delta'')$.

3. Case: $\delta = \mathsf{add}\ (v_i, \lambda_i, \mu_i)$:

$$V' = V \cup \{v_i\} \quad \lambda'(v) = \begin{cases} \lambda(v) & \text{if } v \neq v_i, \\ \lambda_i & \text{if } v = v_i, \end{cases} \quad \mu'(v) = \begin{cases} \mu(v) & \text{if } v \neq v_i, \\ \mu_i & \text{if } v = v_i. \end{cases}$$

4. Case: $\delta = \mathsf{add}\ (v_i, p_{ij}, v_j)$: $E' = E \cup \{(v_i, p_{ij}, v_j)\}$.
5. Case: $\delta = \mathsf{rem}\ v$: $V' = V \setminus \{v\}$.
6. Case: $\delta = \mathsf{rem}\ e$: $E' = E \setminus \{e\}$.
7. Case: $\delta = \mathsf{mod}\ \lambda(v_i)\ \mathsf{by}\ \lambda_j$: $\lambda'(v) = \begin{cases} \lambda(v) & \text{if } v \neq v_i, \\ \lambda_j & \text{if } v = v_i, \end{cases}$
8. Case: $\delta = \mathsf{mod}\ \mu(v_i)\ \mathsf{by}\ \mu_j$: $\mu'(v) = \begin{cases} \mu(v) & \text{if } v \neq v_i, \\ \mu_j & \text{if } v = v_i, \end{cases}$
9. Case: $\delta = \mathsf{mod}\ (v_i, p_{ij}, v_j)\ \mathsf{by}\ q_{ij}$: $E' = \Big(E \setminus \{(v_i, p_{ij}, v_j)\}\Big) \cup \{(v_i, q_{ij}, v_j)\}$.

Case 1 means that the application of an empty delta does not change the $PAAD^J$. Case 2 describes the application of delta operations as a recursive function, which finishes after applying all delta operations after the other. Cases 3 and 4 cover the *add* delta operations for nodes and edges. Removing a node v and an edge e from a $PAAD^J$ just removes them from the sets of nodes V and edges E. The modification of λ and μ is straightforward, since we are dealing with mathematical functions. An exception is the modification of a probability of an edge which is encoded by removing the existing edge and adding the desired edge [KST14].

Figure 5.8 illustrates the known **Stamp** variant of the PPU as a $PAAD^J$. The **Optimized** variant can be obtained by a simple modification of the service rate in the $Crane_M$ component with $\mu = 0.04$. Thus, it slightly increases the performance in terms of an increased processing of workpieces. The formal specification of the respective deltas is given in Example 5.1. Again, these three variants serve as application examples for our efficient performance analysis, before a large-scale evaluation is conducted in Section 5.5.

Example 5.1: PPU Workflow Delta with Performance Annotations

δ-Operation from **Basic** $\Rightarrow$ **Stamp**:

$$\begin{aligned} \delta_S = \{&add(Crane_M, 0.00, 0.03), add(Stamp, 0.0, 0.3), \\ &mod(Stack, 1.0, Crane)\ by\ 0.67,\ add(Stack, 0.33, Crane_M), \\ &add(Crane_M, 1.0, Stamp),\ add(Stamp, 1.0, Slide) \end{aligned}$$

δ-Operation from **Stamp** $\Rightarrow$ **Optimized**:

$$\delta_O = mod\ \mu(Crane_M)\ by\ 0.04$$

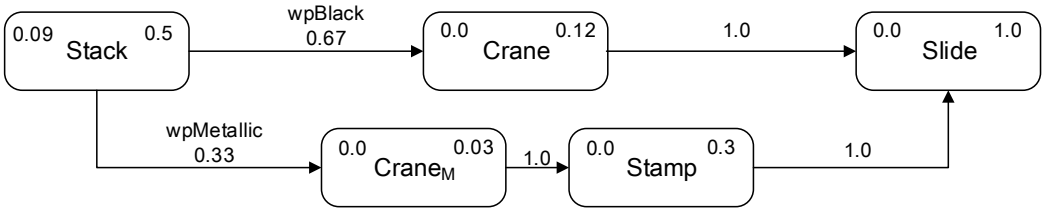


Figure 5.8: **Stamp** variant of the PPU with performance annotations.

At this point, we are able to fully model a product line such as the PPU with our $PAAD^J$s and perform a product-based performance analysis. However, solving the traffic equations over and over again for each variant in isolation is inefficient and computationally expensive. Thus, we generate an artificial super-variant from our core model and the associated set of deltas. We refer to this super-variant as *150%-model*. This is an over-saturated variant representing the whole product line which, in general, does not correspond to a concrete variant of interest to the modeler [TAK+14, KST14]. Each element that is changed by a delta is represented with a parameter instead of its concrete values in the 150%-model. A *150%-model* of the PPU contains the three variants **Basic**, **Stamp** and **Optimized** in one large model.

We now consider a *core* $PAAD^J$ and a set of deltas Δ. We define the 150%-model as a special kind of $PAAD^J$, which has all nodes and transitions that are introduced or modified by each $\delta \in \Delta$. As discussed, in general the 150%-model is not a valid $PAAD^J$ variant, but it contains all the information to retrieve a variant resulting from the application of any $\delta \in \Delta$. The origin of a node or transition from the core model or a specific delta, where it is added, modified or removed, is traced by means of a labeling function $\mathcal{L}$, defined as follows [KST14].

Definition 5.14: 150%-Model

Let $PAAD_c^J = (V_c, E_c, \lambda_c, \mu_c)$ be the *core model* and Δ be a set of consistent and applicable deltas. Let $L = \{C\} \cup \{\underline{\delta}, \underline{!\delta} \mid \delta \in \Delta\}$, with $C \notin \Delta$, be the set of labels. The *150%-model* is $PAAD_{150}^J = (V_{150}, E_{150}, \lambda_{150}, \mu_{150}, \mathcal{L})$, where:

$V_{150} = V_c \cup \{v \mid \exists \delta \in \Delta : \mathsf{add}\ (v, \lambda_i, \mu_i) \in \delta\}$,
$E_{150} = E_c \cup \{(v_i, p_{ij}, v_j) \mid \exists \delta : \mathsf{add}\ (v_i, p_{ij}, v_j) \in \delta \vee \mathsf{mod}\ (v_i, q_{ij}, v_j)\ \mathsf{by}\ p_{ij} \in \delta\}$,

λ_{150} and μ_{150} are partial functions of $V_{150} \times L$ defined as

$$\lambda_{150} : V_{150} \times L \to \mathbb{R}_{\geq 0},\ \lambda_{150}(v,l) = \begin{cases} \lambda_c(v) & \text{if } l = C \wedge v \in V_c, \\ \lambda_i & \text{if } l = \underline{\delta} \wedge \mathsf{add}\ (v, \lambda_i, \mu_i) \in \delta, \\ \lambda_j & \text{if } l = \underline{\delta} \wedge \mathsf{mod}\ \lambda(v_i)\ \mathsf{by}\ \lambda_j \in \delta, \\ 0 & \text{if } l = \underline{!\delta} \wedge \mathsf{rem}\ v \in \delta, \end{cases}$$

$$\mu_{150} : V_{150} \times L \to \mathbb{R}_{\geq 0},\ \mu_{150}(v,l) = \begin{cases} \mu_c(v) & \text{if } l = C \wedge v \in V_c, \\ \mu_i & \text{if } l = \underline{\delta} \wedge \mathsf{add}\ (v, \lambda_i, \mu_i) \in \delta, \\ \mu_j & \text{if } l = \underline{\delta} \wedge \mathsf{mod}\ \mu(v_i)\ \mathsf{by}\ \mu_j \in \delta, \\ 0 & \text{if } l = \underline{!\delta} \wedge \mathsf{rem}\ v \in \delta, \end{cases}$$

and $\mathcal{L}$ is the *labeling function* defined as

$$\mathcal{L} : V_{150} \cup E_{150} \to 2^L,$$

$$\mathcal{L}(v) = \begin{cases} C & \text{if } v \in V_c, \\ \emptyset & \text{otherwise,} \end{cases} \cup \{\underline{\delta} \mid \mathsf{add}\ (v, \lambda_i, \mu_i) \in \delta\} \cup \{\underline{!\delta} \mid \mathsf{rem}\ v \in \delta\}.$$

$$\mathcal{L}(e) = \begin{cases} C & \text{if } e \in E_c, \\ \emptyset & \text{otherwise,} \end{cases} \cup \{\underline{\delta} \mid \mathsf{add}\ e \in \delta\} \cup \{\underline{!\delta} \mid \mathsf{rem}\ e \in \delta\}$$
$$\cup \{\underline{\delta} \mid \mathsf{mod}\ (v_i, q_{ij}, v_j)\ \mathsf{by}\ p_{ij} \in \delta \wedge e = (v_i, p_{ij}, v_j)\}$$
$$\cup \{\underline{!\delta} \mid \mathsf{mod}\ (v_i, q_{ij}, v_j)\ \mathsf{by}\ p_{ij} \in \delta \wedge e = (v_i, q_{ij}, v_j)\}.$$

In order to construct the 150%-model, we consider all nodes V_{150} which are either part of the core $PAAD^J$ or are added in a delta. The set of edges E_{150} contains all edges from the core and all edges added by a delta. Since the probability of an existing edge can be modified by a delta, we add an edge with the new probability to the 150%-model. As a result, we have an edge with the previous probability and an edge with the modified one in the 150%-model. The domain of the functions λ_{150} and μ_{150} are pairs, where the first element indicates the node or edge that is labeled and the second pair specifies a delta label. The functions map onto the concrete value of the rate that the element has in that specific delta. Finally, the labeling function $\mathcal{L}$ is necessary in order to identify the core and the original $PAAD^J$ variants in order to map the results of a performance analysis to the individual $PAAD^J$ variants. Nodes have three possible labels: C means that the node is part of the core $PAAD^J$. Since deltas add or remove nodes, we use $\underline{\delta}$ to denote addition, and $\underline{!\delta}$ for removal. The labeling of edges is done in a similar way. The 150%-model of the PPU is shown in Fig. 5.9. Nodes and edges occur only in the core model, e.g., $\mathcal{L}(v) = \{C\}$, are

marked with solid lines; otherwise, dashed lines are used. The labels of the nodes are shown within the nodes in the bottom part. The delta to generate the **Stamp** variant from the core is denoted as δ_S and the delta to generate the **Optimized** variant is derive as δ_O in Fig. 5.9.

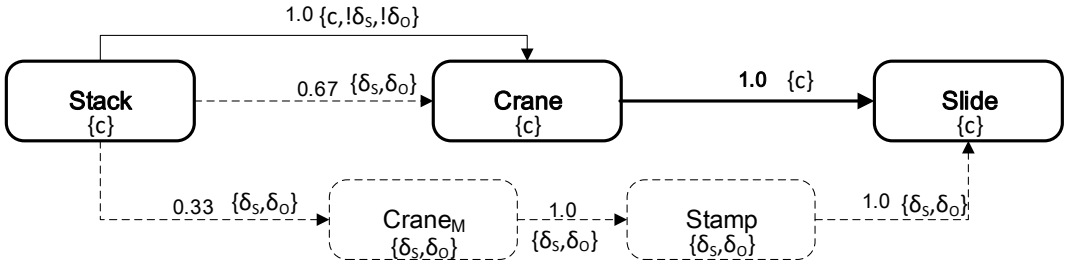


Figure 5.9: 150%-model of the PPU containing the **Basic**, **Stamp** and **Optimized** variants.

The following definition considers a *projection*, i.e., how to obtain a concrete $PAAD^J$ obtained by applying a delta to a core model, from a 150%-model.

Definition 5.15: Projection

Let $PAAD^J_{150}$ be a 150%-model obtained from a core $PAAD^J_c$ through a set of deltas Δ. Then its projection to $\delta \in \Delta$, denoted by $project(PAAD^J_{150}, \delta)$, is given by $PAAD^J_p = (V_p, E_p, \lambda_p, \mu_p)$, where

$$V_p = \{v \in V_{150} : C \in \mathcal{L}(v) \vee \underline{\delta} \in \mathcal{L}(v)\} \setminus \{v \ : !\underline{\delta} \in \mathcal{L}(v)\},$$
$$E_p = \{e \in E_{150} : C \in \mathcal{L}(v) \vee \underline{\delta} \in \mathcal{L}(v)\} \setminus \{e : !\underline{\delta} \in \mathcal{L}(e)\},$$
$$\lambda_p : V_p \to \mathbb{R}_{\geq 0}, \quad \text{with}$$
$$\lambda_p(v) = \begin{cases} \lambda_{150}(v, \underline{\delta}) & \text{if it is defined,} \\ \lambda_{150}(v, C) & \text{otherwise,} \end{cases}$$
$$\mu_p : V_p \to \mathbb{R}_{>0}, \quad \text{with}$$
$$\mu_p(v) = \begin{cases} \mu_{150}(v, \underline{\delta}) & \text{if it is defined,} \\ \mu_{150}(v, C) & \text{otherwise.} \end{cases}$$

In order to complete the cycle, we prove the following theorem, which shows that projecting a 150%-model onto a delta is equivalent to applying the same delta to the core model.

Theorem 5.1: Core to Delta equals 150%-Model to Delta

For an applicable and consistent delta δ, let

$$(V_a, E_a, \lambda_a, \mu_a) = apply(PAAD_c^J, \delta)$$
$$and$$
$$(V_p, E_p, \lambda_p, \mu_p) = project(PAAD_{150}^J, \delta).$$

Then it holds that $(V_a, E_a, \lambda_a, \mu_a) = (V_p, E_p, \lambda_p, \mu_p)$.

Proof. We show that $v \in V_a \iff v \in V_p$. Suppose $v \in V_a$. Then it is because $v \in V_c$ or add $(v_i, \lambda_i, \mu_i) \in \delta$. In all cases it holds that $v \in V_p$. Viceversa, assume $v \in V_p$. This means that $C \in \mathcal{L}(v)$ or $\underline{\delta} \in \mathcal{L}(v)$ and that $\underline{!\delta} \notin \mathcal{L}(v)$; that is $v \in V_c$ or add $(v_i, \lambda_i, \mu_i) \in \delta$, and rem $v \notin \delta$. Thus it holds $v \in V_a$. Similar arguments hold for showing that $e \in E_a \iff e \in E_p$. Comparing λ_{150}, and λ' in Definition 5.13, we infer that $\lambda_a(v) = \lambda_p(v)$ and $\mu_a(v) = \mu_p(v)$, for any $v \in V_a = V_p$.

5.3.2 Family-Product-Based Performance Analysis

Given the 150%-model, we propose an efficient family-product-based performance analysis (cf. Chapter 2, Def. 2.8). The concept is based on symbolic computation yielding an expression for the complete family of products and not just a single variant. This is achieved by solving the traffic equations for the 150%-model. Afterwards, the solution still contains parameters for each element that is changed by a delta as defined within the 150%-model. By plugging in the concrete values of an individual variant into this symbolic expression, we get the actual performance metric for that variant. This allows the re-use of the same symbolic expression *for all variants*, unlike the solution with product-based evaluation in which the traffic equations have to be solved for each variant from scratch.

We now discuss the symbolic evaluation of the 150%-model in more detail. In essence, we take a 150%-model and associate a symbolic variable to each element that is varied in at least one delta. Let $\mathcal{S}$ denote the set of all symbolic variables, whose elements are indicated by a superscript '$*$'.

Definition 5.16: Family-based Evaluation

Let $PAAD_{150}^J$ be a 150% model. The *family-product-based evaluation* is given by the solution of

$$(I - P_s^T)\gamma_s = \lambda_s, \tag{5.11}$$

where:

$$\lambda_s : V_{150} \to \mathbb{R} \cup \mathcal{S},\ \lambda_s(v) = \begin{cases} \lambda_{150}(v, C) & \text{if } \nexists l \in L \setminus \{C\} : \lambda_{150}(v, l) \text{ is defined.} \\ \lambda_v^* & \text{otherwise,} \end{cases}$$

$$\mu_s : V_{150} \to \mathbb{R} \cup \mathcal{S},\ \mu_s(v) = \begin{cases} \mu_{150}(v, C) & \text{if } \nexists l \in L \setminus \{C\} : \mu_{150}(v, l) \text{ is defined.} \\ \mu_v^* & \text{otherwise,} \end{cases}$$

$$P_s = (p^s_{v,v'})_{v,v' \in V_{150}},\ p^s_{v,v'} = \begin{cases} q & \text{if } \exists e = (v, q, v') \in E_{150} \wedge \mathcal{L}(e) = \{C\}, \\ 0 & \text{if } \nexists e = (v, q, v') \in E_{150}, \\ p^*_{v,v'} & \text{otherwise.} \end{cases}$$

Informally, $\lambda_s(v)$ (and similarly, $\mu_s(v)$) treat all parameters that are changed by at least one delta operation as symbolic. Else, the parameter is simply the concrete value assigned in the core model. Concrete probabilities $p_{v,v'}$ are assigned when two nodes are associated *only in the core model*, or when they are not associated at all, i.e., probability of 0. Otherwise, the symbolic variable $p^*_{v,v'}$ is used [KST14].

For an illustrative explanation, let us consider again our example of the PPU in Fig. 5.9. By Definition 5.16, the family-product-based evaluation is

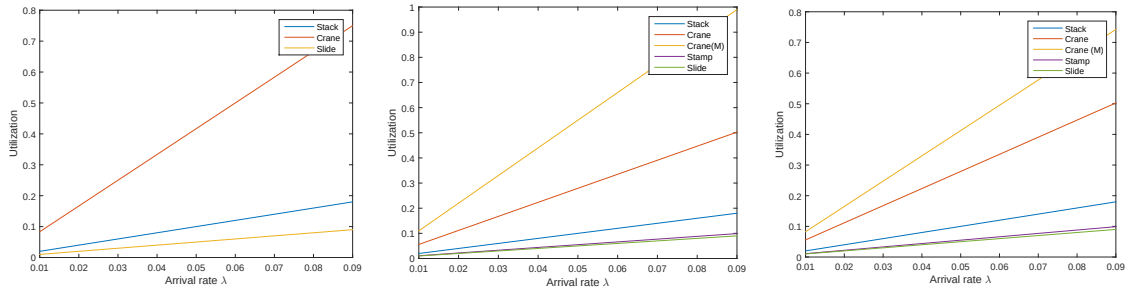
$$P_s = \begin{bmatrix} 0.0 & p^*_{St,C} & p^*_{St,C_M} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & p^*_{C_M,S} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & p^*_{S,Sl} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \quad \lambda_s = \begin{bmatrix} \lambda^*_{St} \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \quad \mu_s = \begin{bmatrix} 0.5 \\ 0.12 \\ \mu^*_{C_M} \\ \mu^*_S \\ 1.0 \end{bmatrix} \tag{5.12}$$

where $\mathcal{S} = \{p^*_{St,C},\ p^*_{St,C_M},\ p^*_{C_M,S},\ p^*_{S,Sl},\ \mu^*_{C_M}, \mu^*_S,\ \lambda^*_{St}\}$. The vectors are ordered as follows: Stack, Crane, Crane$_M$, Stamp and Slide. This notation is also used for the routing probability matrix. Note that the arrival rate at the stack component also is represented by a symbol, which we use in the following to compute the utilization of the PPU for several arrival rates in a family-product-based manner. Thus, the utilization will be given by the following symbolic expression:

$$Util = \Big[2 * \lambda^*_{St}, \frac{25 * \lambda^*_{St} * p^*_{St,C}}{3}, \frac{\lambda^*_{St} * p^*_{St,C_M}}{\mu^*_{C_M}}, \frac{10 * \lambda^*_{St} * p^*_{C_M,S} * p^*_{St,C_M}}{\mu^*_S}, \\ \lambda^*_{St} * p^*_{St,C} + \lambda^*_{St} * p^*_{C_M,S} * p^*_{S,Sl} * p^*_{St,C_M}\Big] \tag{5.13}$$

We observe that plugging in the concrete parameters of the **Basic** variant with $p^*_{St,C} = 1.0$, $p^*_{St,C_M} = 0.0$, $p^*_{C_M,S} = 0.0$, $p^*_{S,Sl} = 0.0$, $\mu^*_{C_M} = 0.0$, $\mu^*_S = 0.0$ and $\lambda^*_{St} = 0.09$ in the symbolic expressions yields an identical result as the previous product-based evaluation of the **Basic** variant. Again, we compute the stack with

18%, the crane with 75% and the slide with 9% utilization. The utilization or any other performance metric of all variants can be computed by evaluating *the same symbolic expression*, assigning the appropriate concrete values related to each variant. Instead, as discussed, product-based evaluation does not allow the re-use of any computation because the solution is based on numerical matrix inversion. For instance, we can efficiently compute the utilization behavior of all components under the aspect of an increasing arrival rate at the stack as depicted in Fig. 5.10. We can observe a linear growth in the components utilization for all variants. Considering the **Stamp** variant, the crane is at maximum capacity for an arrival rate of 0.09. As expected, the **Optimized** variant performs slightly better compared to the **Stamp** one. However, we can identify the crane as the system bottleneck for all cases.



(a) **Basic**. (b) **Stamp**. (c) **Optimized**.

Figure 5.10: Utilization based on varying arrival rates.

We are now left with showing that the symbolic evaluation with the appropriate concrete parameters of a variant always corresponds to the product-based evaluation, i.e., the non-symbolic numerical analysis of a single variant in isolation. The following definition *concretizes* a 150% model with respect to a delta δ, i.e., it isolates the elements of the 150% model that are relevant for δ.

Definition 5.17: Concretization

Let $PAAD^J_{150}$ be a 150% model from a core $PAAD^J_c$ with a set of deltas Δ and with symbolic FB evaluation (5.11). A *concretization* of $PAAD^J_{150}$ *for* $\delta \in \Delta$ is given by $(I - P_k^T)\gamma_k = \lambda_k$, where

$$\lambda_k = (\lambda_k(v))_{v \in V_{150}},\ \lambda_k(v) = \begin{cases} \lambda_{150}(v, \underline{\delta}) & \text{if defined,} \\ \lambda_{150}(v, C) & \text{if defined and } \lambda_{150}(v, \underline{\delta}) \text{ is not defined,} \\ 0 & \text{otherwise,} \end{cases}$$

$$P_k = (p^k_{v,v'})_{v,v' \in V_{150}},\ p^k_{v,v'} = \begin{cases} p^s_{v,v'} & \text{if } p^s_{v,v'} \notin \mathcal{S}, \\ p & \text{if } \exists e = (v, p, v') \in E_{150} \wedge \underline{\delta} \in \mathcal{L}(e), \\ 0 & \text{otherwise.} \end{cases}$$

The concretization creates a system of equations with the size of the 150%-model. However, all symbols from the symbolic expressions are now replaced by the concrete values for a specific variant. Contrary, the projection derives a concrete variant from the 150%-model first and inserts the specific values afterwards. Thus, we get a system of equations with a much smaller size as it includes just one variant. Nevertheless, if we compute the final performance metrics, these metrics should be identical. The next theorem is the desired, crucial result of consistency. It states that the family-product-based symbolic solution, *restricted* to those nodes that are in the variant given by $apply(PAAD_c^J, \delta)$, corresponds to the product-based evaluation of $apply(PAAD_c^J, \delta)$ itself [KST14].

Theorem 5.2: Consistency

Let $(I - P_a^T)\gamma_a = \lambda_a$ denote the product-based evaluation of $(V_a, E_a, \lambda_a, \mu_a) = apply(PAAD_c^J, \delta)$, for $\delta \in \Delta$, and let $(I - P_k^T)\gamma_k = \lambda_k$ be the concretization of the 150% model $PAAD_{150}$ for δ. Furthermore, we define

$$V^\delta = \left\{ v \in V_{150} : \left(C \in \mathcal{L}(v) \wedge \underline{!\delta} \notin \mathcal{L}(v) \right) \vee \underline{\delta} \in \mathcal{L}(v) \right\}.$$

It holds that i) $V^\delta = V^a$ and ii) $\gamma_a(v) = \gamma_k(v)$ and $\mu_a(v) = \mu_k(v)$, for all $v \in V^\delta$ [KST14].

Proof. Suppose first that $v \in V^\delta$ and assume toward a contradiction that $v \notin V_a$. If $C \in \mathcal{L}(v) \wedge \underline{!\delta} \notin \mathcal{L}(v)$ then this implies that $v \in V_c$ and $\mathsf{rem}\ v \notin \delta$, which in turns implies that $v \in V_a$, a contradiction. Similarly, if $\underline{\delta} \in \mathcal{L}(v)$ then $\mathsf{add}\ (v, \lambda_i, \mu_i) \in \delta$, which must again imply $v \in V_a$, because by consistency of deltas, an added node cannot be removed in the same δ. The direction $v \in V_a \implies v \in V^\delta$ can be proven similarly.

Let $p^k_{v,v'}$ denote an element of P_k for all $v, v' \in V_{150}$, and, similarly, let $p^a_{v,v'}$ denote an element of P_a. First, we prove that $p^k_{v,v'} = p^a_{v,v'}$ for all $v, v' \in V^\delta$. We do so by case distinction on $p^k_{v,v'}$. Suppose that $p^k_{v,v} = p^s_{v,v'}$ and $p^s_{v,v'} \notin \mathcal{S}$. This implies that $\exists e = (v, p^s_{v,v'}, v') \in E_{150}, \mathcal{L}(e) = \{C\}$. Therefore, by contradiction it is possible to see that none of the operations $\mathsf{add}\ e$, $\mathsf{rem}\ e$, and $\mathsf{mod}\ e\ \mathsf{by}\ q$ is in δ. Since $\mathcal{L}(e) = \{C\}$ implies $e \in E_c$ and e is not altered by δ in any way, then $e \in E_a$, thus $p^k_{v,v'} = p^s_{v,v'}$. Now suppose that $p^k_{v,v'} = p$ because $\exists e = (v, p, v') \in E_{150} \wedge \underline{\delta} \in \mathcal{L}(e)$. This must necessarily imply that either $\mathsf{add}\ e \in \delta$ or $\mathsf{mod}\ (v, q, v')\ \mathsf{by}\ p \in \delta$. In both cases it holds that $(v, p, v') \in E_a$ and therefore $p^a_{v,v'} = p = p^k_{v,v'}$. We now turn to the case $p^k_{v,v'} = 0$. This may be due to two cases: a) $\nexists e = (v, p, v') \in E_{150}$, for any p, or b) $\exists e = (v, p, v') \in E_{150}$, $\underline{\delta} \notin \mathcal{L}(e)$ and $C \notin \mathcal{L}(e)$. If a) holds, then clearly $(v, p, v') \notin E_a$ for any p a fortiori, hence $p^a_{v,v'} = 0$. If b) holds e may either not be considered at

all in the operations in δ (although, e.g., it may hold that $\underline{\delta'} \in \mathcal{L}(e)$, for $\delta \neq \delta'$); or $\underline{!\delta} \in \mathcal{L}(e)$. This would imply that $\mathsf{rem}\ (v, p, v') \in \delta$, hence $p^k_{v,v'} = 0$ also in this case.

Now, we show that $p^k_{v,v'} = 0$ for $v \in V^\delta$ and $v' \notin V^\delta$. Suppose toward a contradiction that $p^k_{v,v'} > 0$ for some v'. Then, again by case distinction: if $p^k_{v,v'} = p^s_{v,v'}$, with $p^s_{v,v'} \notin \mathcal{S}$ then $\exists e = (v, p^k_{v,v'}, v') \in E_{150}, \mathcal{L}(e) = \{C\}$. This implies that $C \in \mathcal{L}(v)$ and $C \in \mathcal{L}(v')$. Now we show that $\underline{!\delta} \notin \mathcal{L}(u)$, for $u = v, v'$. Suppose that $\underline{!\delta} \in \mathcal{L}(u)$, then $\mathsf{rem}\ u \in \delta$. This implies, by consistency, that all connected edges should be removed in the same δ, hence $\underline{!\delta} \in \mathcal{L}(e)$, a contradiction. For the other case, we consider $\exists e = (v, p, v') \in E_{150} \wedge \underline{\delta} \in \mathcal{L}(e)$. This implies that either $\mathsf{add}\ e \in \delta$ or $\mathsf{mod}\ (v, q, v')\ \mathsf{by}\ p \in \delta$, but in both cases it must hold by consistency (cannot add an edge between vertices that are not there or that will be removed, or cannot modify an edge between vertices that will be removed) that $v, v' \in V^\delta$, a contradiction. For the same reason, it holds that $p^k_{v,v'} = 0$ for $v \notin V^\delta$ and $v' \in V^\delta$. Analogous arguments can be used to show the remaining equivalences.

Thus, overall, the system $(I - P^T_k)\gamma_k = \lambda_k$ can be written in the block structure

$$\left(\begin{array}{c|c} (I - P^\delta_k)^T & 0 \\ \hline 0 & (I - P^{\delta^c}_k)^T \end{array}\right) \left(\begin{array}{c} \gamma^\delta_k \\ \hline \gamma^{\delta^c}_k \end{array}\right) = \left(\begin{array}{c} \lambda^\delta_k \\ \hline \lambda^{\delta^c}_k \end{array}\right),$$

where the upper elements refer to the nodes in V^δ and the bottom elements refer to the nodes in $V^{\delta^c} = V_{150} \setminus V^\delta$. Therefore, we conclude that

$$\gamma^\delta_k = (I - P^{\delta^T}_k)^{-1}\lambda^\delta_k = (I - P^T_a)^{-1}\lambda_a = \gamma_a.$$

A summary of the results presented thus far is shown in Figure 5.11 and considering the two problems of complexity from the beginning of this chapter, we can make the following observations:

- *Model-specific complexity:* This problem is basically avoided by observing that the models of our interest, i.e. Jackson networks with a single class of customers and an exponential distributed service time, admit a very efficient product-form solution [Jac63, Ste09].

- *Family-related complexity:* The computation of a symbolic expression representing the complete product line provides a solution for this aspect. Thus, we are able to re-use computations completely independent from the number of variants.

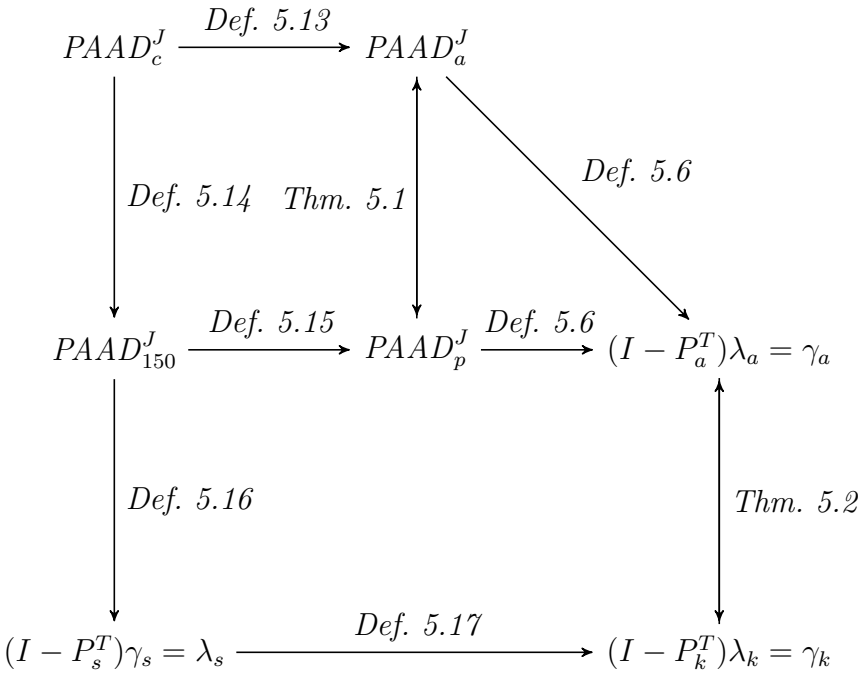


Figure 5.11: Summary of results. Single-headed arrows illustrate how to derive objects from other objects using the definitions presented in this thesis. Double-headed arrows indicate the results of consistency between the 150% and the single products obtained by a δ application.

5.4 Creating the 150%-Variability Model for Coxian Queueing Networks

Similar to the previous section, we now propose an efficient family-product-based performance analysis for the second class of networks with Coxian distributed service times. Analyzing a $PAAD^C$ involves solving a system of linear equations (5.9). Again, we exploit commonalities in a variability model in order to perform this calculation only once on the 150%-model. As variability modeling techniques, we use once again delta modeling and the overall procedure is similar to the one for Jackson networks. We require the typical well-formedness conditions for the $PAAD^C$s ensuring that no dangling nodes are present, or that the sum of all outgoing probabilities from a node is equal to 1 (cf. Def. 5.12 and Section 4.2.2). The main differences to Jackson networks consist of the new performance annotations, i.e., multiple servers, Coxian rate and probability vectors and the number of customers, which have to be captured by delta operations. Additionally, we need to solve a different type of traffic equations to generate the 150%-model.

5.4.1 Application of Delta Modeling

First of all, we formally define the notion of delta for $PAAD^C$s with Coxian service distributions, multiple servers and a closed QN. To this end, we first recall that any vector can be seen as a function on natural numbers. For instance, an $x \in \mathbb{R}^2$ is a function $x : \{1,2\} \to \mathbb{R}$ such that $x = (x(1), x(2))$. This view becomes useful whenever the nodes $\mathcal{V}$ of a $PAAD^C$ do not form a set $\{1,\dots,n\}$ but an arbitrary finite subset of $\mathbb{N}$. This can easily happen, for instance, when nodes are removed or replaced by other nodes. With this in mind, in the following we interpret vectors as functions and identify S^i, C^i, μ^i and p^i by $S(i)$, $C(i)$, $\mu(i)$ and $p(i)$, respectively. Again, for simplicity, in this thesis we assume that each variant is induced by a single delta operation (cf. Section 4.2.2).

Definition 5.18: *PAAD*^{C} Deltas

A $PAAD^C$ delta (based on Def. 5.7) is a set of delta operations $\delta \subseteq Op$, where

$$
\begin{aligned}
Op = \{&\mathsf{add}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i)) \mid i \in \mathbb{N}, \mathcal{C}(i) \in \mathbb{N}_{\geq 0},\\
&\mathcal{S}(i) \in \mathbb{N}_{>0} \text{ and } \mu(i), p(i) \text{ define a Coxian distribution}\}\\
&\cup \{\mathsf{rem}\ e \mid e \in \mathbb{N} \times (0;1] \times \mathbb{N}\}\\
&\cup \{\mathsf{add}\ (i, r_{ij}, j) \mid i, j \in \mathbb{N}, r_{ij} > 0\} \cup \{\mathsf{rem}\ i \mid i \in \mathbb{N}\}\\
&\cup \{\mathsf{mod}\ (i, r_{ij}, j)\ \mathsf{by}\ \tilde{r}_{ij} \mid (i, r_{ij}, j) \in \mathcal{E}, \tilde{r}_{ij} > 0\}\\
&\cup \{\mathsf{mod}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i))\ \mathsf{by}\\
&\quad (\tilde{\mathcal{C}}(i), \tilde{\mathcal{S}}(i), \tilde{\mu}(i), \tilde{p}(i)) \mid \tilde{\mathcal{C}}(i) \in \mathbb{N}_{\geq 0}, \tilde{\mathcal{S}}(i) \in \mathbb{N}_{>0}\\
&\qquad \text{and } \tilde{\mu}(i), \tilde{p}(i) \text{ define a Coxian distribution}\}
\end{aligned}
$$

Up to this point, it is not ensured that the application of a delta also leads to a well-formed $PAAD^C$ variant (cf. Def. 5.8). We require that any delta must be *applicable* and *consistent* fulfilling the conditions described in Section 4.2. Additionally, the underlying routing matrix has to remain irreducible, implying in particular that the probabilities of the outgoing edges of a node have to sum up to one.

The following definition formalizes how to obtain an arbitrary variant through application of a delta to a $PAAD^C$.

Definition 5.19: *PAAD*^{C} Delta Application

The application of an applicable and consistent delta $\delta \subseteq Op$ to a $PAAD^C = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{S}, \mu, p)$ is defined by the function $PAAD^{C'} = apply(PAAD^C, \delta)$, where $PAAD^{C'} = (\mathcal{V}', \mathcal{E}', \mathcal{C}', \mathcal{S}', \mu', p')$. It is recursively defined as follows.

1. Case $\delta = \emptyset$: $PAAD^{C'} = PAAD^C$.
2. Case: $\delta = \delta' \cup \delta'' \wedge \delta', \delta'' \in Op$: $PAAD^{C'} = apply(apply(PAAD^C, \delta'), \delta'')$.

3. Case: $\delta = \mathsf{add}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i))$:

$$\mathcal{C}' = \mathcal{C} \cup \{(i, \mathcal{C}(i))\} \qquad \mathcal{S}' = \mathcal{S} \cup \{(i, \mathcal{S}(i))\}$$
$$\mu' = \mu \cup \{(i, \mu(i))\} \qquad p' = p \cup \{(i, p(i))\}$$
$$\mathcal{V}' = \mathcal{V} \cup \{i\}$$

4. Case: $\delta = \mathsf{add}\ (i, r_{ij}, j)$: $\mathcal{E}' = \mathcal{E} \cup \{(i, r_{ij}, j)\}$.
5. Case: $\delta = \mathsf{rem}\ i$: $\mathcal{V}' = \mathcal{V} \setminus \{i\}$.
6. Case: $\delta = \mathsf{rem}\ e$: $\mathcal{E}' = \mathcal{E} \setminus \{e\}$.
7. Case: $\delta = \mathsf{mod}\ (i, r_{ij}, j)\ \mathsf{by}\ \tilde{r}_{ij}$: $\mathcal{E}' = \Big(\mathcal{E} \setminus \{(i, r_{ij}, j)\}\Big) \cup \{(i, \tilde{r}_{ij}, j)\}$.
8. Case: $\delta = \mathsf{mod}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i))$
   $\mathsf{by}\ (\tilde{\mathcal{C}}(i), \tilde{\mathcal{S}}(i), \tilde{\mu}(i), \tilde{p}(i))$:

$$\mathcal{C}' = \Big(\mathcal{C} \setminus \{(i, \mathcal{C}(i))\}\Big) \cup \{(i, \tilde{\mathcal{C}}(i))\}$$
$$\mathcal{S}' = \Big(\mathcal{S} \setminus \{(i, \mathcal{S}(i))\}\Big) \cup \{(i, \tilde{\mathcal{S}}(i))\}$$
$$\mu' = \Big(\mu \setminus \{(i, \mu(i))\}\Big) \cup \{(i, \tilde{\mu}(i))\}$$
$$p' = \Big(p \setminus \{(i, p(i))\}\Big) \cup \{(i, \tilde{p}(i))\}$$

Case 1 and 2 are identical compared to the application of deltas to Jackson networks (cf. Def. 5.13). In case 3 lies the first major difference as a service station is now equipped with two additional parameters, i.e., number of customers $\mathcal{C}(i)$ and servers $\mathcal{S}(i)$. In addition, the external arrival rate in Jackson networks is exchanged with the Coxian probability vector $p(i)$. A parameter for the service rate is present in Jackson and Coxian QN. Cases 4-7 also have semantics that can be found in a Jackson network. However, instead of modifying the individual rates at a service station, we now always modify the complete service station to keep the number of delta operations low for developers [KTTS15, KTTS17].

For instance, let us consider the following delta δ_{DP} applied to the core $PAAD_c^C$ of Fig. 5.5.

$$\begin{aligned}\delta_{DP} = \{&\mathsf{rem}\ (\mathbf{P1}, {}^{4}\!/\!_{5}, \mathbf{W1}), \mathsf{rem}\ (\mathbf{W2}, 1, \mathbf{P1}),\\ &\mathsf{rem}\ (\mathbf{D}, 1, \mathbf{T}), \mathsf{rem}\ (\mathbf{P1}, {}^{19}\!/\!_{100}, \mathbf{D}),\\ &\mathsf{add}\ (\mathbf{L1}, 0, 10, ({}^{1}\!/\!_{2}, 1), ({}^{1}\!/\!_{2}, 1)),\\ &\mathsf{add}\ (\mathbf{P5}, 0, 2, (20000, 10000), ({}^{1}\!/\!_{2}, 1)),\\ &\mathsf{add}\ (\mathbf{L2}, 0, 10, ({}^{1}\!/\!_{2}, 1), ({}^{1}\!/\!_{2}, 1)),\\ &\mathsf{add}\ (\mathbf{P1}, {}^{99}\!/\!_{100}, \mathbf{L1}), \mathsf{add}\ (\mathbf{L1}, 1, \mathbf{P5}),\\ &\mathsf{add}\ (\mathbf{P5}, {}^{2}\!/\!_{10}, \mathbf{L2}), \mathsf{add}\ (\mathbf{L2}, {}^{1}\!/\!_{2}, \mathbf{D}), \mathsf{add}\ (\mathbf{D}, 1, \mathbf{L2}), \mathsf{add}\ (\mathbf{L2}, {}^{1}\!/\!_{2}, \mathbf{P5}),\\ &\mathsf{add}\ (\mathbf{P5}, {}^{1}\!/\!_{10}, \mathbf{L1}), \mathsf{add}\ (\mathbf{P5}, {}^{7}\!/\!_{10}, \mathbf{W1}), \mathsf{add}\ (\mathbf{W2}, 1, \mathbf{P5})\}.\end{aligned}$$

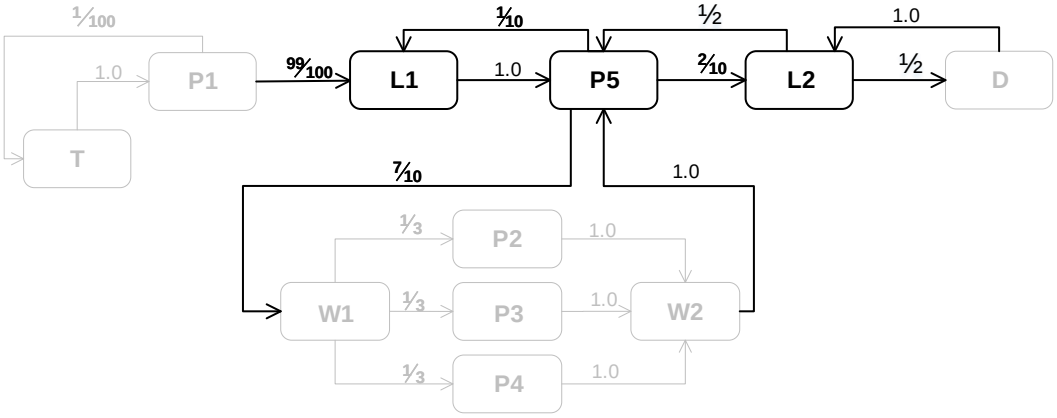


Figure 5.12: $PAAD^C$ variant obtained by applying δ_{DP}. **L1** and **L2** are new delay stations representing a LAN.

Its overall intent is to change the IRS to a *distributed* variant. This variant can answer more user requests before it reaches its maximum capacity by enabling research in a local network instead of always relying on the wide area network [CM02]. This is realized by the following sequence of basic operations: removal of the edges between the **P1** and both **W1** and **W2**, as well as all edges to station **D**. Then, we add two delay nodes to the network with **L1**, **L2**, and a new CPU (**P5**). This can be shown to implement the modifications presented in [CM02, Fig. 16].

The addition or removal of an edge is syntactically defined by three parameters with *source*, *probability* and *target*. Adding a node requires five parameters with *name*, *#-customers*, *#-servers* followed by the service time distribution vectors. Removing a node can simply be achieved by referring to its *name*. Overall, this leads to the addition of nine new transitions. Figure 5.12 shows the resulting $PAAD^C$ after applying δ_{DP} to the core $PAAD^C_c$. Deep black parts indicate the changes performed by the delta. Table 5.2 summarizes the performance annotations of this variant.

Table 5.2: Performance annotations for the $PAAD^C$ variant of Fig. 5.12 obtained by applying delta δ_{DP}.

Name	*Customers*	*Servers*	*Rates*	*Probabilities*
T	10	10	(1,2)	(1/2,1)
D	0	1	(4000, 2000)	(1/2,1)
P1	0	4	(20000, 10000)	(1/2,1)
P2	0	4	(20000, 10000)	(1/2,1)
P3	0	4	(20000, 10000)	(1/2,1)
P4	0	4	(20000, 10000)	(1/2,1)
P5	0	2	(20000, 10000)	(1/2,1)
L1	0	10	(1/2,1)	(1/2,1)
L2	0	10	(1/2,1)	(1/2,1)
W1	0	10	(1,2)	(1/2,1)
W2	0	10	(1,2)	(1/2,1)

To show an example of a delta obtained through modification of existing $PAAD^C$ elements, we consider δ_F, which represents a failure of **P4** compensated by an increase of the servers at stations **P2** and **P3** by one:

$$\begin{aligned}\delta_F = \{&\mathsf{rem}\ \mathbf{P4}, \mathsf{rem}\ (\mathbf{W1}, 1/3, \mathbf{P4}), \mathsf{rem}\ (\mathbf{P4}, 1, \mathbf{W2}),\\ &\mathsf{mod}\ (\mathbf{W1}, 1/3, \mathbf{P2})\ \mathsf{by}\ 1/2,\\ &\mathsf{mod}\ (\mathbf{W1}, 1/3, \mathbf{P3})\ \mathsf{by}\ 1/2,\\ &\mathsf{mod}\ (\mathbf{P2}, 0, 5, (20000, 10000), (1/2, 1)),\\ &\mathsf{mod}\ (\mathbf{P3}, 0, 5, (20000, 10000), (1/2, 1))\}.\end{aligned}$$

Figure 5.13 and Table 5.3 show the corresponding QN model and its performance annotations, respectively.

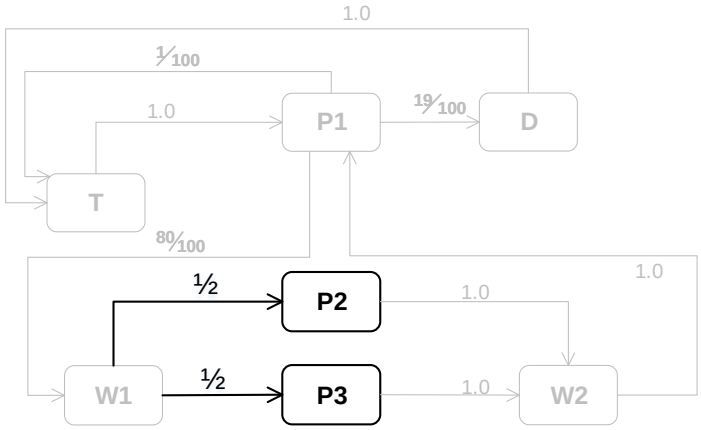


Figure 5.13: $PAAD^C$ variant obtained by applying δ_F to the core. Again, deep black parts indicate the changed model elements.

Table 5.3: Performance annotations of the $PAAD^C$ variant of Fig. 5.13 obtained by applying delta δ_F.

Name	*Customers*	*Servers*	*Rates*	*Probabilities*
T	10	10	(1,2)	(1/2,1)
D	0	1	(4000, 2000)	(1/2,1)
P1	0	4	(20000, 10000)	(1/2,1)
P2	0	5	(20000, 10000)	(1/2,1)
P3	0	5	(20000, 10000)	(1/2,1)
W1	0	10	(1,2)	(1/2,1)
W2	0	10	(1,2)	(1/2,1)

We remark that typical studies [ETCS14, GF99] such as what-if scenarios can be encoded in this delta-based framework. For instance, understanding the performance when the server multiplicity at station $\mathbf{T}$ varies from 1 to 10 can be encoded with 10 different deltas, i.e., $\delta_k = \{\mathsf{mod}\ (\mathbf{T}, 10, k, (1,2), (1/2,1))\}$, with $1 \leq k \leq 10$. Notably, our framework also allows both structural and parametric changes.

5.4.2 Family-Product-Based Performance Analysis

Let us consider a core $PAAD_c^C$ with a set of deltas Δ. As stated earlier, the 150%-model is an over-saturated $PAAD^C$ consisting of all nodes and edges that are added or modified by some delta $\delta \in \Delta$. Although it is not a valid $PAAD^C$ variant in general, we have the necessary information to derive a specific variant of the considered system. This information is stored in the 150%-model with the help of a labeling function ϕ.

Definition 5.20: Creating the 150%-model

Let $PAAD_c^C = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{C}_c, \mathcal{S}_c, \mu_c, p_c)$ be the *core model* and Δ be a set of consistent and applicable deltas. Let $\mathcal{L} = \{c\} \cup \{\underline{\delta}, \underline{!\delta} \mid \delta \in \Delta\}$ be the set of labels.

The *150%-model* is

$$PAAD_{150}^C = (\mathcal{V}_{150}, \mathcal{E}_{150}, \mathcal{C}_{150}, \mathcal{S}_{150}, \mu_{150}, p_{150}, \phi),$$

where:

$$\begin{aligned}\mathcal{V}_{150} &= \mathcal{V}_c \cup \{i \mid \exists \delta \in \Delta : \mathsf{add}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i)) \in \delta\},\\ \mathcal{E}_{150} &= \mathcal{E}_c \cup \{(i, r_{ij}, j) \mid \exists \delta : \mathsf{add}\ (i, r_{ij}, j) \in \delta \vee\\ &\qquad\qquad \mathsf{mod}\ (i, r_{ij}, j)\ \mathsf{by}\ r'_{ij} \in \delta\}.\end{aligned}$$

For any $X \in \{\mathcal{C}, \mathcal{S}, \mu, p\}$, we define the partial functions

$$X_{150} : \mathcal{V}_{150} \times \mathcal{L} \to \mathbb{R}_{\geq 0}, X_{150}(i,l) = \begin{cases} X_c(i) & \text{if } l = c \wedge i \in \mathcal{V}_c,\\ X(i) & \text{if } l = \underline{\delta} \wedge \mathsf{add}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i)) \in \delta,\\ X'(i) & \text{if } l = \underline{\delta} \wedge \mathsf{mod}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i))\\ & \qquad \mathsf{by}\ (\mathcal{C}'(i), \mathcal{S}'(i), \mu'(i), p'(i)) \in \delta,\\ 0 & \text{if } l = \underline{!\delta} \wedge \mathsf{rem}\ i \in \delta, \end{cases}$$

At last, ϕ is the *labeling function* defined as

$$\phi : \mathcal{V}_{150} \cup \mathcal{E}_{150} \to 2^{\mathcal{L}},$$

$$\phi(i) = \begin{cases} c & \text{if } i \in \mathcal{V}_c, \\ \emptyset & \text{otherwise,} \end{cases} \cup \{\underline{\delta} \mid \mathsf{add}\ (i, \mathcal{C}(i), \mathcal{S}(i), \mu(i), p(i)) \in \delta\} \cup \{\underline{!\delta} \mid \mathsf{rem}\ i \in \delta\}.$$

$$\phi(e) = \begin{cases} c & \text{if } e \in \mathcal{E}_c, \\ \emptyset & \text{otherwise,} \end{cases} \cup \{\underline{\delta} \mid \mathsf{add}\ e \in \delta\} \cup \{\underline{!\delta} \mid \mathsf{rem}\ e \in \delta\} \cup \{\underline{\delta} \mid \mathsf{mod}\ (i, r_{ij}, j)\ \mathsf{by}\ r'_{ij} \in \delta \wedge e = (i, r'_{ij}, j)\} \cup \{\underline{!\delta} \mid \mathsf{mod}\ (i, r_{ij}, j)\ \mathsf{by}\ r'_{ij} \in \delta \wedge e = (i, r_{ij}, j)\}.$$

The 150%-model of $PAAD^C$s is based on all nodes $\mathcal{V}_{150}$ and edges $\mathcal{E}_{150}$ that are either part of the core model or added by a delta. For each modified probability, one edge with the new probability is added to the 150%-model to ensure that no information is lost. The domain of the partial function X_{150}, with $X \in \{\mathcal{C}, \mathcal{S}, \mu, p\}$, is a pair where the first element indicates the node or edge that is labeled, and the second parameter specifies a delta label. In addition, the labeling function ϕ keeps track of the deltas that affect a node or an edge. For instance, $\phi(e) = \{c, \underline{!\delta}\}$ says that the edge e is present in the core model and is removed by δ. Instead, $\phi(e) = \{\delta\}$ expresses the fact that e is not present in the core model and is added by δ. The main difference to the construction of the 150%-model in Jackson networks (cf. Def. 5.14) is to keep track of the additional performance annotations in the service stations, i.e., number of customers and servers as well as the Coxian vectors. If any modification in a service station occurs, we store all of the service station parameters in the 150%-model and not just the changed value. In Jackson networks it was possible to modify arrival or service rates separately without specifying the other parameters of the service station [KTTS15, KTTS17].

The complexity of constructing a 150%-model is linear with the number of deltas. Given our labeling function, we can iterate through all deltas and add the information about their specific operations to the core model, which ultimately results in the 150%-model. A prerequisite is that the set of deltas is consistent, which should be ensured beforehand. However, the full construction process can be automated and does not necessarily require expert knowledge.

Figure 5.14 shows a graphical representation of the 150%-model which subsumes the core model in Fig. 5.5 as well as the two variants originated from δ_{DP} and

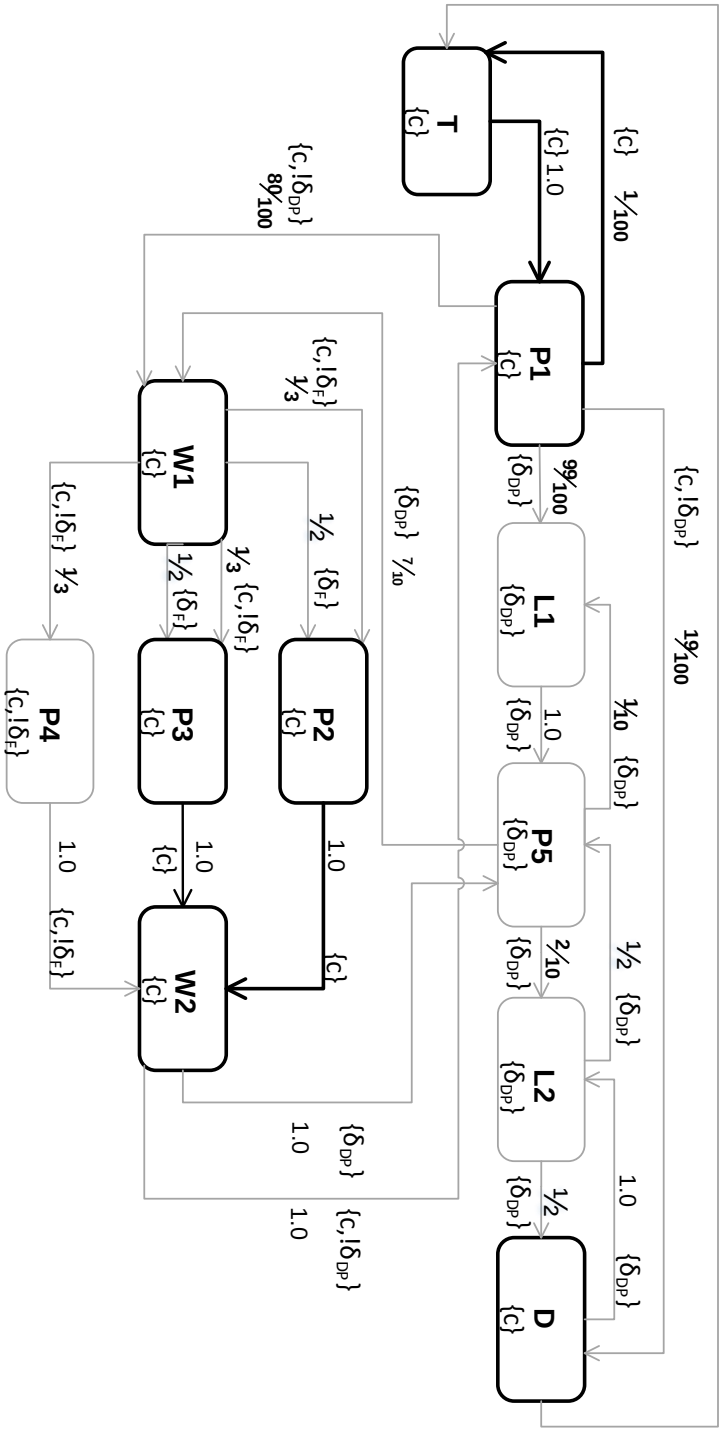


Figure 5.14: 150% PAD^C for the core of Fig. 5.5 with δ_{DP} and δ_F.

δ_F. Notationally, nodes and edges that do not occur in all variants are drawn with light-gray lines. Elements that do occur in every variant are marked with solid deep-black lines. Labels shown at the bottom-middle of each node indicate the presence conditions as given by the labeling function. For instance, label c refers to the core model while label δ_{DP} in node **P5** means that the node is only present in the variant obtained by applying δ_{DP}. A similar labeling is used for edges. We omit showing the X_{150} functions here, since they would only summarize the contents of Tables 5.1–5.3.

We now discuss the symbolic evaluation of the 150% model in detail. Again, the *concretization* allows us to obtain a specific variant from the 150%-model. Its definition is similar to the concretization for a 150%-model in case of Jackson networks (cf. Def. 5.17) and therefore omitted at this point [KTTS15, KTTS17]. The next theorem states that a concretization $PAAD^C_\delta$ of a 150%-model with respect to a given delta coincides with the application of the delta to the core model $PAAD^C_c$. This statement will be needed later to relate the product-based evaluation to the family-product-based evaluation. Its proof is similar to the proof of Theorem 5.2.

Theorem 5.3: Consistency

Let $PAAD^C_{150}$ be a 150%-model and $\delta \in \Delta$ and $PAAD^C_\delta = \mathit{apply}(PAAD^C_c, \delta)$. Let R_δ be the routing probability matrix of $PAAD^C_\delta$ and $R^\dagger_\delta \zeta = \zeta$ be its traffic equations. Denote by ζ_δ the unique solution of the traffic equations when $\zeta^1_\delta = 1$. Then, the product-based evaluation of $\mathit{apply}(PAAD^C_c, \delta)$ is given by T^i_δ, for $i \in \mathcal{V}_\delta$, where

$$T^i_\delta = \begin{cases} \zeta^i_\delta (\sum_{j=1}^n \mathbb{E}^j_\delta \zeta^j_\delta)^{-1} \sum_{j=1}^n \mathcal{C}_\delta(j) & \text{if (5.10)}, \\ \zeta^i_\delta \min \Big\{ \mathcal{S}_\delta(j) (\zeta^j \mathbb{E}^j_\delta)^{-1} \mid \\ \qquad \text{with } j \in \mathcal{V}_{150} \text{ such that } \mathcal{S}_\delta(j) < \infty \Big\} & \text{else} \end{cases}$$

As in the case of Jackson networks, we can write the traffic equations of the Coxian QN for the 150% $PAAD^C$ model. Instead of solving the traffic equations $R^\dagger_\delta \zeta_\delta = \zeta_\delta$ for each variant $PAAD^C_\delta$ separately, we provide a solution for all variants of the family. The key idea relies on the introduction of a routing matrix R_s on the set $\mathcal{V}_{150}$, with entries that are either constants or symbolic variables. In particular, to incorporate variability, all elements that are added, removed, or modified by at least one delta are replaced by symbolic expressions. Theorem 5.3 guarantees that the symbolic solution of the traffic equations evaluated with the concrete values of a given delta provides the correct performance estimates [KTTS15, KTTS17].

Theorem 5.4: Family-product-based Evaluation

Let $PAAD^C_{150}$ be a 150%-model and $\delta \in \Delta$ and $R_s = (r^s_{i,j})_{i,j \in \mathcal{V}_{150}}$ denote the 150% routing matrix, where

$$r^s_{i,j} = \begin{cases} r & \text{if } \exists e = (i, r, j) \in \mathcal{E}_{150} \wedge \mathcal{L}(e) = \{c\}, \\ 0 & \text{if } \nexists e = (i, r, j) \in \mathcal{E}_{150}, \\ r^*_{i,j} & \text{otherwise.} \end{cases}$$

Let r^* denote the vector of symbolic variables $r^*_{i,j}$ present in R_s. Let ζ_s denote the *symbolic* solution of

$$R^\dagger_s \zeta_s = \zeta_s, \qquad \text{with } \zeta^1_s = 1. \tag{5.14}$$

Then, it holds that $\zeta^i_s = \zeta^i_\delta$ for all $i \in \mathcal{V}_\delta$ if r^* is such that

$$r^*_{i,j} = \begin{cases} (R_\delta)_{i,j} & \text{if } i, j \in \mathcal{V}_\delta, \\ 0 & \text{otherwise.} \end{cases}$$

Its proof is available in [KTTS17].

Let us illustrate Theorem 5.4 using the 150%-model of Fig. 5.14. The symbolic routing matrix R_s is given in Fig. 5.15. The theorem states that evaluating ζ_s with the probability values of a delta δ coincides with the product-based solution of the concrete variant obtained by applying δ to the core model.

$$R_s = \begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1/100 & 0.0 & r^*_{P1,D} & r^*_{P1,W1} & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{P1,L1} & 0.0 & 0.0 \\ r^*_{D,T} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{D,L2} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{W1,P2} & r^*_{W1,P3} & r^*_{W1,P4} & 0.0 & 0.0 & 0.0 \\ 0.0 & r^*_{W2,P1} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{W2,P5} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & r^*_{P4,W2} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{L1,P5} & 0.0 \\ 0.0 & 0.0 & 0.0 & r^*_{P5,W1} & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{P5,L1} & 0.0 & r^*_{P5,L2} \\ 0.0 & 0.0 & r^*_{L2,D} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & r^*_{L2,P5} & 0.0 \end{pmatrix}$$

Figure 5.15: Symbolic routing matrix

For instance, instantiating R_s with the values of δ_F yields the matrix in Fig. 5.16.

$$R_s(r_{\delta_F}) = \begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1/100 & 0.0 & 19/100 & 80/100 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1/2 & 1/2 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Figure 5.16: Routing matrix of the 150%-model with values for δ_F

Instead, applying δ_F to the core model yields the routing matrix:

$$R_{\delta_F} = \begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1/100 & 0.0 & 19/100 & 80/100 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.00 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1/2 & 1/2 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{pmatrix}$$

The elimination of rows $8, 9, 10$ and 11 and columns $8, 9, 10$ and 11 from $R_s(r_{\delta_F})$ gives rise to R_{δ_F}. More importantly, eliminating the coordinates 8-11 of $\zeta_s(r_{\delta_F})$ will give us the unique solution of the traffic equations $R^{\dagger}_{\delta_F}\zeta_{\delta_F} = \zeta_{\delta_F}$. A similar result can be obtained for δ_{DP}.

Overall, by symbolically solving the equation (5.14), we have a solution for the whole family. This underpins the usefulness of our technique when the number of variants is large as discussed next. Considering the two problems of complexity from the beginning of this chapter, we can make the following observations:

- *Model-specific complexity:* This problem is circumvented by using an ODE approximation for the CTMC model to provide an efficient solution in form of traffic equations. Each service center has its own set of ODEs and adding a new service center results only in a linear number of new equations. The complexity of the performance analysis solely depends on the network topology.
- *Family-related complexity:* The computation of a symbolic expression representing the complete SPL also provides a solution in this case. We are able to re-use computations completely independent from the number of variants.

5.5 Evaluation

The evaluation is divided into two main parts. After a description of our prototypical implementation in our multi-perspective modeling framework, we focus first on the evaluation of Jackson networks and second on the evaluation of QN with a Coxian

service time distribution. The considered case studies are comprised of artificially generated and realistic performance models to compare the product-based and the family-product-based approaches in large-scale SPLs. In particular, we investigate the following research questions:

- **RQ 5.1**: *What is the computational benefit of the family-product-based performance analysis in case of Jackson networks?* Our proposed technique is only meaningful, if there is a significant difference in terms of computation times.
- **RQ 5.2**: *How does this benefit change for the more powerful QN type containing Coxian distibutions?* Similar to the previous research question. However, due to extensive changes in the mathematical background, we have to investigate the potential computational benefit again.
- **RQ 5.3**: *What is the impact of variability on computation times in case of Coxian QN?* We are especially interested in the distribution of symbols throughout the routing probability matrix and a possible negative impact for the computational benefits of the family-product-based analysis.

5.5.1 Implementation

We extended the implementation of our multi-perspective modeling approach (cf. Chapter 4) with the proposed performance annotations. The graphical representation of the Jackson networks was already presented in the previous sections 5.2.1 and 5.3. Again, users can choose between the graphical and textual representation in our multi-perspective modeling approach to create a Jackson network. The following Listing presents the **Basic** PPU with performance annotations in our textual DSLs. Its semantics are identical to Fig. 5.4.

```
ActivityDiagram PPU: Basic {
    Tasks{
        Start ( type= Initial, label= "Start" )
        Stack ( type= Action,  label= "Stack",
                    arrival= "0.09", service= "0.5" )
        Crane ( type= Action,  label= "Crane",
                    arrival= "0.00", service= "0.12" )
        Slide ( type= Action,  label= "Slide",
                    arrival= "0.00", service= "1.0" )
        End   ( type= Final,   label= "End"   )}
    ControlFlows{
        flow0 = Start => Stack
        flow1 = Stack => Crane (probability="1.0")
        flow2 = Crane => Slide (probability="1.0")
        flow3 = Slide => End}}
```

Listing 5.1: $PAAD^J$ of the PPU: Textual.

Furthermore, a product-based performance analysis is directly possible within our framework. The mathematical computations are executed with the Apache Commons Math library.[1] However, for the family-product-based analysis an export of the routing probability matrix and the values for service and arrival rates to Matlab[2] is necessary. Matlab has an efficient symbolic solver as part of the *Symbolic Math Toolbox*. In case of the Coxian networks, the product-based and the family-product-based analyses are both completely performed in Matlab. For instance, the symbolic expressions for the IRS running example are created with the following listing of Matlab code in which zeta (ζ) is the unique solution of the system of equations (5.14) and E represents the mean service time $\mathbb{E}$.

```
%Define Variables, Matrix, Servers, Customers, and E
...
%Prepare Symbolic Solving
zeta = [l0 l1 l2 l3 l4 l5 l6 l7 l8 l9 l10]';
A = [(R')*([l0 l1 l2 l3 l4 l5 l6 l7 l8 l9 l10]')]-zeta;
A(1,:) = [];
Rs = subs(A, l0, 1);

%Solve system of equations symbolically
S = solve(Rs,l1 ,l2 ,l3 ,l4 ,l5, l6, l7, l8, l9, l10);

%Buffer solutions
zeta = [1 S.l1 S.l2 S.l3 S.l4 S.l5 S.l6 S.l7 S.l8 S.l9
    S.l10];
Mult = E * zeta';

%Create symbolic expressions
for k = 1:11
  Rv(k) = (E(k)*zeta(k)*Customers)/Mult;   %Condition
  Tt(k) = (zeta(k)*Customers)/Mult;         %TRUE
  Mi(k) = Server(k)/(zeta(k)*E(k));         %FALSE
end
...
```

Listing 5.2: Extract from the Matlab-File.

After the definition and initialization of the necessary variables, we have to prepare the matrix equations for the symbolic solver. Line 4 defines the vector, which later contains the unique solution of ζ. In line 5, we perform a basic matrix operation

[1] http://commons.apache.org/proper/commons-math/
[2] https://de.mathworks.com/products/matlab.html

by bringing the system of equations $R_s^\dagger \zeta_s = \zeta_s$ into the form $R_s^\dagger \zeta_s - \zeta_s = 0$ and store the result in A. This is followed by removing the first row, i.e., the first linear equation, from the system of equations (Line 6). Removing one equation is a necessary prerequisite to find a solution, but we are not restricted to delete the first one as any choice is valid. We already know the result of this removed equation because we set its solution $l0$ to 1 (cf. Theorem 5.4). Line 7 substitutes all appearances of $l0$ with 1 in the system of equations. Finally, we can symbolically solve the equations in Line 10. The unique solution is saved in ζ (Lines 13-14) and we perform a multiplication of the ζ with the mean service times, which we need in the coming calculations. Lines 18-22 create the final symbolic expressions for each of the eleven nodes in the IRS. We create the bottleneck condition in line 19, the expressions for the throughputs if no bottleneck is present (Line 20), and the expressions for the throughputs if a bottleneck is present in the network in line 21. Afterwards, we can plug-in the concrete values of an individual variant, check the condition for all nodes and compute the performance metrics. In general, the process is similar for Jackson networks with respect to the different system of equations and formulas to determine performance metrics as described in Section 5.2.1. Exemplary symbolic expressions for the PPU modeled as a Jackson network were already given in equation 5.13. The final symbolic expressions for the IRS as a Coxian network consist of several hundred characters making a representation in this thesis not reasonable.

5.5.2 Evaluation of Jackson Networks

We compared the family-product-based symbolic analysis against the product-based approach, where each configuration is solved separately for a given concrete set of parameter values. The focus lies on computation times in order to answer **(RQ 5.1)**.

Our experimental set-up was as follows. We considered randomly generated 150% $PAAD^J$ models with different numbers of nodes and variables for their symbolic evaluation, i.e., the set of elements in $\mathcal{S}$. In detail, let n denote the total number of nodes in the family-product-based evaluation, i.e., $n = |V_{150}|$, p be the number of variables in the routing probability matrix (i.e., the number of symbols in P_s), m the number of variables in the service rates (the symbols in the range of the function in μ_s), and g the number of variables in the exogenous arrival rates (the symbols in the range of λ_s). For any given choice of (n, p, m, g), all the other parameters were randomly generated, with the service rates drawn uniformly at random in $[1.0; 20.0]$, and the arrival rates in the range $[0.0; 3.0]$. For instance, the symbolic evaluation of the PPU in (5.12) corresponds to a configuration $(n, p, m, g) = (5, 4, 1, 2)$.

For each tuple (n, p, m, g), we considered 200 randomly generated variants, which we analyzed with both the product-based and the family-product-based approach.

In essence, we randomly generated 200 tuples, each of length $p+m+g$, corresponding to a specific instantiation of the symbolic parameters. For the family-product-based approach, each tuple was used to evaluate a symbolic expression for the average queue length; for the product-based approach instead, the parameters were used to solve the system of equations for each variant. We measured the execution times for both performance analyses, repeated 5 times in order to reduce the noise in the measurements. The tests were implemented in Matlab using the *Symbolic Math Toolbox* for the family-product-based evaluation, and the built-in functions for the solutions of systems of linear equations for the product-based evaluation. The measurements were conducted on a machine with an Intel Core i7 2.66 GHz with 8 GB RAM.

Each row in Table 5.4 shows the overall execution times, averaged over the 5 runs, of both analyses across the 200 random variants, which represent the whole family for each configuration (n, p, m, g). We report the average speedup product-based/family-product-based (PB/FB). The last column shows the pre-computation (PC) time taken to symbolically solve equation (5.11). These results allow us to make the following observations [KST14].

- We confirm that for any fixed choice of p, m, and g that we considered, larger values of n make a family-product-based analysis increasingly more convenient, with speedups up to over 2000 times; see, for instance, the configurations $(4, 0, 1, 0)$, $(24, 0, 1, 0)$, $(142, 0, 1, 0)$, and $(302, 0, 1, 0)$. These configuration have a low share of variability with one symbol in the service rates and get the highest benefit from the family-product-based analysis. We can make a similar observation for other configurations with few symbols. This is because of the increasing cost of solving the system of linear equations for the product-based approach, while our proposed analysis solves it symbolically only once.
- For fixed n, varying p, m, and g has an impact on speedup, since the higher the number of variables the larger the symbolic expression derived by the family-product-based approach.
- For fixed n, not all other parameters affect the speedup in the same manner. In particular, compare the two cases $p = m = g = 2$ and $p = g = 0$, $m = 6$, for every given n. Both have the same number of variables (i.e., six), but the latter case consistently enjoys a better speedup. This is because the m variables do not appear in equation (5.1), thus for $m = 6$ the symbolic expressions of the solution of equation (5.1) can be greatly simplified. The m variables will appear in the calculation for the queue lengths L_v.
- The family-product-based evaluation is not always more efficient than the naive approach. In our study, this has occurred in smaller models (i.e., $n = 24$) with relatively high number of variables. In these cases, the symbolic expression turned out to be more difficult to compute than the linear system of equations for which Matlab is well-known to be optimized.

- The pre-computation time behaves well with the number of variables, in particular with respect to the p variables that are used in the solution of equation (5.1).

Overall, the advantage of our family-product-based performance analysis would increase further if we analyze more than 200 variants. However, we did also observe that the number of symbols have an impact on computation times and therefore the benefit of the proposed performance analysis. This aspect needs additional investigation. Nevertheless, we conclude that **(RQ 5.1)** is answered sufficiently with an increasing benefit for large networks and many variants.

Variables				*Runtimes (s)*			
n	p	m	g	*FB*	*PB*	*PB/FB*	*PC*
4	1	0	0	0.011	0.049	4.47	0.545
4	0	1	0	0.004	0.043	10.78	0.185
4	0	0	1	0.009	0.045	4.92	0.190
4	1	1	1	0.011	0.046	4.13	0.214
4	2	2	2	0.014	0.049	3.60	0.319
4	4	4	4	0.020	0.049	2.43	0.310
24	1	0	0	0.011	0.066	5.96	1.141
24	0	1	0	0.004	0.063	14.60	1.124
24	0	0	1	0.011	0.067	6.29	1.152
24	1	1	1	0.013	0.068	5.04	1.244
24	2	2	2	0.016	0.068	4.31	1.100
24	0	6	0	0.007	0.065	9.94	1.004
24	4	4	4	0.099	0.070	0.70	1.904
142	1	0	0	0.016	6.540	397.34	5.425
142	0	1	0	0.005	8.107	1664.28	4.908
142	0	0	1	0.015	10.808	726.41	4.836
142	1	1	1	0.017	10.069	601.51	5.113
142	2	2	2	0.019	10.052	526.50	4.936
142	0	6	0	0.007	7.802	1191.79	5.137
142	4	4	4	0.026	10.985	429.83	4.942
302	1	0	0	0.019	19.186	1007.95	11.026
302	0	1	0	0.006	13.680	2292.46	11.192
302	0	0	1	0.018	13.399	728.73	11.050
302	1	1	1	0.021	19.520	909.12	11.591
302	2	2	2	0.024	19.459	820.30	11.214
302	0	6	0	0.006	13.896	2258.21	11.089
302	4	4	4	0.030	14.879	495.06	11.718

Table 5.4: Results [KST14]. FB=Family-product-based, PB=Product-based.

5.5.3 Evaluation of Networks with Coxian Rates

Again, we first compare the product-based and the family-product-based analyses in terms of computation times. We considered a model of a realistic three-tiered system, *Rubis*, which has been employed as a benchmark application for various evaluation purposes in the software performance community (e.g., [UPS+05, GDK+14, NSG+13]). While in the original model each tier was described by a single station, here we consider a load-balancing version where each tier has a number of parallel stations, similarly to [ITT16]. The goal of a load-balancing system is to distribute workloads across multiple resources, such as disk drives, computers, clusters, and network links. This is useful to optimize the usage of available resources, maximize system throughput, minimize system response time, and avoid overloading individual resources.

Figure 5.17 depicts the smallest model instance that we considered. The *sentry* node is a delay station which collects all requests to the system. Each tier i is made of a dispatcher **Di** which distributes customers to the three parallel servers (i.e., **S1**–**S3** for tier one). This basic structure was used to build larger QNs by increasing the number of servers at each tier by a factor of ten and one hundred, resulting in QNs with 94 and 904 stations, respectively. For a given network size, hereafter denoted by n, we varied the number of symbolic variables. In particular, r denotes the number of symbols in the routing probability matrix, E represents the number of symbols in the mean service times and S refers to the number of server symbols giving us configurations in the form of (n, r, E, S). This allowed us to study models of different size with a realistic topology, going beyond our previous evaluation that considered synthetic Jackson networks.

We remark that software systems with such kinds of variability occur in practice. For example, we can implement different load balancing strategies by leaving entries of the routing probability matrix as symbols. Each variant represents a different choice of the probabilities with which the dispatcher at each tier forwards requests to the processing. In an optimization problem, one would like to find the variant that optimizes a certain performance index. Clearly, the number of possible variants to be analyzed can become huge even for a moderate number of servers and choices of probabilities with 0.01 steps, a granularity typical of real-world load bal-

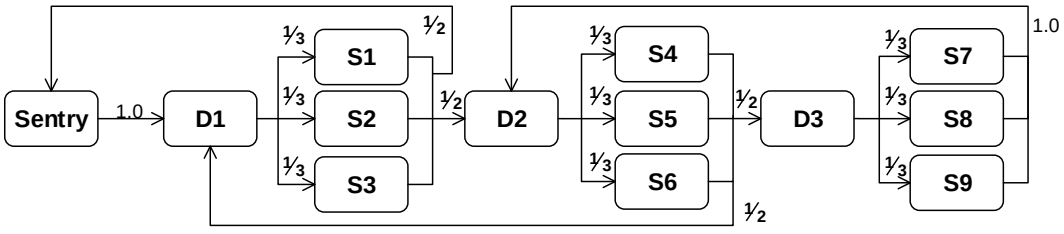


Figure 5.17: An instance of a QN model for the three-tier load balancing system used for the evaluation.

ancers [PKHS10]. Variants with different Coxian rate and probability vectors are useful to perform sensitivity analysis or uncertainty analysis: each variant represents a sample for the parameter space, for instance by fixing mean service times uniformly spaced within a given interval. Finally, variants that contain changes in the number of servers can be useful in cloud computing applications. Here, the number of servers may represent the number of physical or virtual machines that are active. These are typical decision variables in optimization problems used to derive dynamic resource allocation policies [ATZ07].

The comparison index for the evaluation of **(RQ 5.2)** is the *break-even point* (BE), intended as the least number of products for which the overall runtime of the product-based computations exceeds that of the family-product-based analysis (including the symbolic solution of the 150%-model). This is an indicator of the size of the parameter space after which our analysis is computationally more convenient than the product-based analysis. For any given choice of symbolic variables (n, r, E, S), concrete products were generated as follows: the mean service time at each node was sampled uniformly at random from the interval $[1.0; 10.0]$; the server multiplicities were taken uniformly from $[1; 20]$ for each station; the number of customers was 50. The steady-state network throughput was the performance measure of interest in this study.

Again, the evaluation was conducted in Matlab similar to the Jackson networks. Table 5.5 presents the results. For each chosen configuration (n, r, E, S), we measured the average runtime for the evaluation of the steady-state symbolic expression (column *FB*) and of the product-based solution (column *PB*). Column *PC* reports the pre-computation time to symbolically solve the system of linear equations, including the construction of the symbolic expression for the throughput. The last column gives the break-even point, i.e., the number of variants, which we have to analyze before the family-product-based approach is advantageous compared to the product-based approach. We were able to make the following observations based on these results [KTTS15]:

- The break-even point is smaller than 6437 variants across all cases. This promotes the use of a family-product-based analysis — at least for the case study — because it represents a tiny fraction of the potential parameter space that can be explored with the degrees of variability considered in our experiments. Each symbol represents a parameter that takes values in the reals. Thus, even a *single variable* may represent *infinitely many variants*.
- The average evaluation of a symbolic expression is faster than the corresponding product-based analysis, up to one order of magnitude in the case of $n = 904$. In the worst case examined here, our analysis yields a 11% improvement over the product-based one. Hence, the family-product-based analysis analyzes individual variants always faster than the product-based approach.

- The pre-computation times are mostly affected by n and r, the size of the 150%-model and the number of symbols in the routing matrix. The configurations $(13, 13, 0, 0)$ and $(13, 13, 13, 13)$ have the longest pre-computation times for networks of size $n = 13$. A similar observation is possible in case of $n = 94$ and at least the configuration $(904, 13, 13, 13)$. This can be explained by the fact that the main bottleneck of the computations is the solution of the linear system of equations, which requires an explicit inversion of the symbolic routing probability matrix. Other symbols are introduced during the construction of the final symbolic expressions, which takes after the matrix inversion. Thus, symbols in E and S have a lesser impact on computation times.

Variables				*Runtimes (s)*			
n	r	E	S	*FB*	*PB*	*PC*	*BE*
13	6	0	0	0.0009	0.0010	0.39	3997
13	0	6	0	0.0008	0.0011	0.26	875
13	0	0	6	0.0008	0.0010	0.27	1392
13	6	6	6	0.0009	0.0011	0.41	2077
13	13	0	0	0.0009	0.0011	0.74	3744
13	0	13	0	0.0008	0.0010	0.27	1375
13	0	0	13	0.0008	0.0010	0.27	1392
13	13	13	13	0.0009	0.0012	0.48	1612
94	6	0	0	0.0010	0.0019	1.97	2189
94	0	6	0	0.0011	0.0016	1.64	3297
94	0	0	6	0.0011	0.0015	1.58	3975
94	6	6	6	0.0011	0.0016	1.79	3594
94	13	0	0	0.0011	0.0015	2.38	5961
94	0	13	0	0.0011	0.0016	1.67	3343
94	0	0	13	0.0011	0.0015	1.62	4067
94	13	13	13	0.0011	0.0015	2.07	5175
904	6	0	0	0.0020	0.0192	80.54	4683
904	0	6	0	0.0021	0.0223	82.98	4108
904	0	0	6	0.0019	0.0209	79.02	4159
904	6	6	6	0.0018	0.0186	83.98	4999
904	13	0	0	0.0020	0.0216	79.38	4056
904	0	13	0	0.0018	0.0186	85.61	5106
904	0	0	13	0.0019	0.0187	94.96	5636
904	13	13	13	0.0020	0.0178	101.72	6437

Table 5.5: Computation results [KTTS17].

Although the computational advantage of the family-product-based performance analysis decreases compared to the evaluation for Jackson networks, it is still present and significant in case of analyzing several thousand variants. A possible explanation for this decrease can be found in the matrix operations. The product-based analysis of Coxian networks is much faster in Matlab compared to the product-based analysis of Jackson networks. This observation holds even for large networks with 904 nodes. In contrast, the pre-computation time of the family-product-based analysis in Coxian networks stays about the same compared to Jackson networks resulting in the decreased computational benefit of our proposed performance analysis. However, one has to conduct further experiments if this is also the case for other case studies. Thus, we successfully identified the answer to **(RQ 5.2)** with a decrease in the overall efficiency of the family-product-based approach.

Cost of Variability. To understand how the degree of variability impacts the pre-computation time of our family-product-based analysis in Coxian networks, we further studied the structure of the 150%-model. For this we considered synthetic 150%-models. This choice is motivated by the fact that the running example has a rather regular structure such that the symbolic routing probabilities may happen to be in the same row (for the outgoing edges from **D1**, **D2**, and **D3**) or in the same column (for the outgoing edges of the server stations at each tier). This may simplify the computations for the matrix inversion, required to symbolically solve the system of equations in (5.14).

Instead, here we consider an application scenario where the symbolic parameters are more uniformly distributed across the matrix. We show how they affect the runtimes by comparing against a situation where the symbolic parameters are found just in a few rows. Specifically, we studied different configurations of $(16, r, E, S)$ meaning that the network size is fixed at 16 nodes. For each configuration we considered two cases. In the *localized* case the symbolic routing probability matrix R_s was such that the first and second rows contained all symbols, while all remaining rows had only concrete values, drawn uniformly at random such that the sum across any row is equal to one. In the *distributed* case, the symbolic parameters were placed such that the following two conditions were satisfied: i) at least 65% of the rows in R_s contain symbols; and ii) no more than than four symbols can be found in one row. The terms localized and distributed reflect the scattering principle of symbols in the routing matrix.

Table 5.6 reports the runtimes to build the symbolic expressions for the steady-state network throughput. The results demonstrate that the topology of the routing probability matrix has a strong impact on the pre-computation time, with runtimes that are separated by at least two orders of magnitude between the localized and the distributed cases. The influence of the mean service time E and the servers

Variables				*Runtimes (s)*	
n	r	E	S	*Localized*	*Distributed*
16	20	0	0	1.131	113.40
16	20	16	0	1.136	112.71
16	20	0	16	1.129	108.89
16	20	16	16	1.140	113.07
16	24	0	0	2.116	943.12
16	24	16	0	2.114	924.82
16	24	0	16	2.122	940.65
16	24	16	16	2.105	942.54

Table 5.6: Impact of the degree of variability in the routing matrix on the symbolic pre-computation times [KTTS17].

S, instead, is negligible since their related symbolic variables are used only after routing probability matrix is inverted. The number of symbolic variables in the matrix, denoted by r, impacts the runtimes sensibly.

Thus, we conclude that variability has a serious effect on computation times in case of Coxian networks (**RQ 5.3**). We argue that a similar observation is likely for Jackson networks as the symbolic matrix inversion also is the bottleneck for this family-product-based performance analysis. However, in order to verify this statement another experiment is required in the future.

Finally, we remark that the localized and distributed scenarios impacted not only time, but also space. For instance, we registered a peak memory usage of 3.2 GB for the distributed case of the configuration $(16, 24, 16, 16)$. In contrast, the *localized* cases required at most 1 GB RAM. Thus, in case of larger networks with high variability, the available system memory can also become a problem [TAK+14].

5.5.4 Discussion

In this section we, summarize the main limitations of our proposed approach and point out threats to validity.

Limitations. Our starting point is a variability-intensive system that can be represented as a QN. A main limitation of our model is that it only allows to describe "flat" networks. For instance, layering of services (whereby an entity acts both as a customer and as server) is not supported, yet it is a frequent pattern of interaction, e.g., in multi-tier systems. Extending our work to enhanced models such as layered queueing networks, that support such patterns is left for future work. However, we

note that in certain cases it is possible to capture such dynamics still using a flat network, including the QN model of *Rubis* [UPS+05] discussed here.

Another major assumption is that the analysis provides steady-state performance metrics. Lifting the family-product-based analysis to the transient dynamics is a challenging task that we did not address and may be not possible due to the underlying mathematical construct. However, we remark that steady-state metrics are relevant in long-running systems where the time it takes a system to enter the steady-state can be neglected as for the PPU. Indeed, steady-state metrics are most popular in many software performance engineering scenarios [CM02, GF99].

We only support anticipated variability, meaning that all variants must be known beforehand. This is not always the case in SPLE, since a product line may evolve over time, e.g., due to changing requirements and a possible introduction of new variants [MD08]. As a result, the symbolic analysis has to be conducted again afterwards. This unanticipated variability is a challenge, which is left for future work. However, the family-product-based approach can be applied when the model is used at runtime to steer a running system toward an optimal configuration [ITT16]. In this case, it is reasonable to assume that the family does not change during the execution, but different variants may be chosen to control performance. The 150%-model would be computed at each build of the system, but always offline.

Threats to Validity. The evaluation of Jackson networks is based on artificial models in which the symbolic parameters were randomly generated and distributed throughout the routing probability matrix, service and arrival rates. A large-scale SPL from the real world was not analyzed to evaluate the computational benefit of our family-product-based performance analysis. We only applied it to the relative small automation product line of the PPU. Thus, our observations rely on these generated models and we cannot transfer the achieved results with full certainty to real-world SPLs. The next threat deals with the rather low number of symbols, i.e., twelve symbols at most, even in case of large networks with 302 nodes. The computational impact of high variability is not considered in the evaluation of Jackson networks. However, we remark that even a single symbol may represent an infinite number of variants.

The evaluation of networks with Coxian-distributed service times suffers from similar threats. Although we consider a load-balancing system, which is often used as benchmark in the software performance community [UPS+05, GDK+14, NSG+13], the network extension to 94 and 904 nodes is an artificial construction. Again, we generated and distributed the symbols randomly across the routing probability matrix, the mean service time and the number of servers. Hence, we have a realistic example as foundation, but introduced further synthetic models. The threat of a relative low number of symbols also remains. However, in order to mitigate this

threat, we conducted a second experiment in which the share of symbols was much higher compared to the network size. Again, the models used for this evaluation were artificially generated with respect to certain conditions (cf. Section 5.5.3). An application to real SPLs with high variability is missing at this point.

5.6 Related Work

Our contribution follows the line of research in exploiting software models with explicit notions of variability in order to improve the efficiency of the analysis, leveraging commonalities across different variants [TAK+14]. We first create a model representing the complete family of variants in the product line. Next, this model is analyzed giving us the family-based nature in our proposed performance analysis. Afterwards, we derive concrete performance metrics of individual variants representing the product-based part of the analysis. In the terminology defined by **Thüm et al.** [TAK+14] this results in a combined approach called family-product-based analysis (cf. Chapter 2). Furthermore, according to **Thüm et al.** [TAK+14] only a few family-product-based analysis exist in the literature. **Tartler et al.** [TLSSP11] propose a sample-based approach that includes the variability model and prepocessor-based source code to achieve code coverage. This aspect is contrary to typical sample-based approaches, which only consider the variability model. However, **Tartler et al.** [TLSSP11] target bug finding in source code and not properties related to performance. **Shi et al.** [SCD12] propose a family-product-based analysis to cover feature interactions in the product line. Given valid feature combinations specified by the variability model, they derive a dependency graph representing the whole product line. Similar to our performance analysis they use the concept of symbolic execution to create test cases. Afterwards, test cases can be executed to analyze individual products. Non-functional performance properties are not considered by **Shi et al.** [SCD12].

Although not part of the family-product-based approaches, **Siegmund et al.** [SKK+12, SGAK15] also focus on detecting feature interactions. However, the authors aim at predicting the performance impact of specific feature interactions, which allows to derive conclusions about the performance of a customized and generated program. **Siegmund et al.** propose a sample-based strategy for this task in [SKK+12] and combine this work with a machine-learning algorithm in [SGAK15]. **Guo et al.** [GCA+13] follow this line of research by building a performance model from measurements on the real system with a statistical learning approach. The created performance model represents a correlation between feature selections and the resulting impact on performance of the system. We did not explicitly consider feature interactions within our family-product-based performance analysis. While the described approaches rely on system measurements, we focus on models to achieve

a performance prediction. Overall, our proposed performance analysis complements this line of research.

For *qualitative* models and the analysis of functional properties, approaches are the model-checking algorithm of **Classen et al**. [CHS+10, CCS+13], which checks linear temporal logic formulae for a family of models induced by a *featured transition system*. A featured transition system incorporates the information of a feature model in a typical labeled transition system. This combination can be exploited to derive properties for the complete SPL in a family-based fashion, i.e., safety properties. The on-the-fly model checker by **ter Beek al.** [tBFGM15] continues in this direction by proposing a family-based analysis for behavioral models. In particular, this work is based on the fundamental introduction of a featured transition system representing a the whole family of an SPL. The main difference to **Classen et al**. [CHS+10, CCS+13] lies in the choice of the used model checker. **Ghezzi et al.** [GS11] propose an approach based on parametric probabilistic model checking of software product line models as annotated UML sequence diagrams. The annotations depict variations points required to derive specific configurations. Thus, we have a single model for a family of variants. This leads to a symbolic expression that encodes the dependence of certain properties of interest from variables, in a manner which is analogous to ours. However, our work is different in that we consider properties of performance as opposed to reliability/energy.

Filieri et al. [FGT11, FTG15] propose a technique to parametric model checking of discrete-time Markov chains (DTMCs). It consists of a symbolic partial evaluation for the analysis of pCTL formulae, which is followed by an instantiation with the concrete values that may be taken by the "variable" transition probabilities. The focus of **Filieri et al.** [FGT11, FTG15] lies on run-time model checking, while we focus on the design time of a system. Nevertheless, our proposed performance analysis can be used at run-time after the family model is constructed to evaluate possible variants. This is especially beneficial if we encounter a failure of one component, which we have to compensate by re-distributing the workload. Parametric model checking requires the DTMC to be built. In application domains, which focus on *single-user* scenarios—for instance in certain case studies concerning energy consumption [FTG15] and reliability [GS11]—this is feasible because the model does not incur state-space explosion, since the state space essentially depends on the complexity of the user's model. Conversely, single-user scenarios are, to a large extent, uninteresting for performance evaluation because nontrivial interactions arise from many users contending a pool of resources. In this case, the state-space size grows combinatorially with the number of users and resources. In real-world performance-related applications, the enumeration of the state space of the CTMC of a concrete product—let alone a symbolic representation of a family—will likely fail on commodity hardware. Actually, one of our main contributions is that we avoid the CTMC

building altogether by reasoning symbolic over an approximate ODE representation whose size only depends on the network topology. To cope with large-scale DTMCs, **ter Beek et al.** propose the use of statistical model checking [tBLLV15], for a feature-aware process algebra with DTMC semantics that captures a dynamic product line (where the active features can be changed during execution [GH04]). The analysis does not take advantages of the common behaviors among the variants.

There exists research for SPLs that deal specifically with model-based performance analysis. **Tawhid** and **Petriu** consider a UML software product line model [TP11], annotated with performance information, from which they derive performance models as layered queueing networks [FAOW+09]. The problem space variability model is represented by a feature model in which features are annotated with performance-related properties available in the MARTE profile of the UML. The solution space models consist of UML sequence and deployment diagrams, which are also extended with MARTE annotations. The concrete derivation of variants is realized with a model-to-model transformation in the Atlas Transformation Language. A second transformation creates the layered queueing network based on the UML models, which represent the concrete variant. However, the performance analysis is done in a product-by-product fashion, which does not takes advantage of the commonalities in the family. **Exteberria et al.** [ETCS14] consider a performance analysis for feature-aware software designs under uncertainty. In particular, the authors analyze the correlation between a specific feature selection with uncertain parameters and the resulting system performance. Uncertain parameters refer to values that may be unknown during early design phases. Thus, the performance analysis is executed two times for each variant to determine the best and worst case scenarios for these uncertain parameters. A feature model is used as variability model and connected to an architectural model embedding variable features. The variants are generated with the SPLConquerer tool [SRK+12]. Contrary to our family-product-based performance analysis, each variant is still solved in isolation.

Happe et al. [HKR11b] propose a compositional reasoning approach in the *Palladio* framework. It is based on component models with performance annotations and incorporates various analysis tools. For instance, software architects can get immediate feedback on bottlenecks in the system, which is similar to our family-product-based performance analysis in Coxian networks. Nevertheless, Palladio requires a component model, while we operate on behavioral models of the software system. Finally, we are most closely related to research that was conducted parallel to this thesis by **Incerto et. al.** [ITT16]. The authors consider self-adaptive systems as systems with high variability, which are able to change their configuration on-the-fly in case of critical events during run-time. **Incerto et. al.** [ITT16] pursue the goal to find a new configuration satisfying all defined quality-of-service requirements, i.e., throughput, utilization and response time. Similar to our pro-

posed performance analysis, a single QN model is created for the complete family of configurations and solved in a symbolic way. While we use this symbolic solution to directly compute performance metrics for individual variants, **Incerto et. al.** encode the model and the quality of service constraints as SMT problem. Afterwards, the Z3 SMT solver is used to devise a feasible configuration for the system.

5.7 Chapter Summary and Future Work

In this chapter, we have presented an efficient family-product-based performance analysis for SPLs. The analysis is based on UML activity diagrams, which are enhanced with performance annotations. By interpreting these software models as queueing networks, we were able to perform a product-based performance analysis and calculate performance metrics, e.g., throughput, for each variant in isolation. Similar to the contribution in Chapter 4, we manage variability through the application of delta modeling and use it to construct a 150%-model representing the complete product line. A single analysis of this family model enables us to efficiently derive performance metrics for all variants.

Overall, we considered two different types of queuing networks that can be encoded with the respective performance annotations as UML activity diagram. The family-product-based analysis was first introduced for Jackson networks. In Jackson networks, we annotate a service station with the external arrival rate and a service rate. A service station in Coxian networks is annotated with the number of customers, the number of servers, a Coxian service rate vector and a Coxian probability vector. Edges are labeled with a routing probability in both network types. Each performance value that is changed by a delta is represented as a symbol in the 150%-model. Solving the equations of a 150%-model for a given network type presents us with symbolic expressions. The performance metrics of an individual variant are obtained by inserting the concrete values of this variant. The procedure is similar for both network types as they only differ in the equations, which we have to solve. The evaluation has shown a significant benefit in computation times for our family-product-based analysis compared to a product-based one in all cases.

In future work, it should be tested if a family-product-based analysis is applicable to more expressive formalisms. For instance, by removing the limitation of the lack of support for fork/join synchronization barrier, and extend it to layered queueing networks. The efficient analysis of transient behavior, i.e., the time before we enter the steady-state, is still an open point and should be considered in the future. With respect to the development of an SPL, the most crucial problem which we have to solve is the management of unanticipated variability, i.e., evolution of the SPL. An incremental performance analysis may hold the key for an efficient solution. Finally, the proposed approach can be used as a foundation to formulate an optimization problem similar to work done by Incerto et. al. [ITT16].

6 Conclusion

Software product line development is an important task in modern software engineering. The rising amount of variability in software systems threatens single system development strategies and calls for product line methods [PBvdL05, CE05]. Reusability is a key aspect in this paradigm to reduce redundant tasks as much as possible for developers. Many approaches have already been proposed to support the development of an SPL [PBvdL05, CE05, BSRC10, TAK+14]. However, as we have motivated in the introduction of this thesis, there is a lack of adequate product line approaches concerning the automation domain. Variability management is a rather new challenge for developers of an automation system, but gets increasingly more important with a rising software share in such systems [BBO+12, FLVH15, VHFF+15, Vya13].

6.1 Discussion

In this section, we recall the research questions defined in the introduction of this thesis and discuss answers based on our observations. We also take a closer look at an applicability of the contributions to systems beyond the automation domain.

> ***RQ1:*** *How can we reuse and adapt existing variability modeling approaches to deal with the interdisciplinary nature of automation systems in the problem space?*

We use multiple interrelated feature models to fully express an interdisciplinary product line in the problem space. The idea is adapted from work conducted in the topics of feature model composition [RSTS11, HTH13] and multi software product lines [HGR12, BRN+13]. An initial idea for automation systems was already presented by the developers of the PPU in [FLVH15]. However, we significantly extended the existing approach in two directions. First of all, we applied a feature model slicing technique onto the interrelated feature models to identify and highlight possible dependencies between the individual engineering domains and their feature models. We took advantage of an efficient slicing algorithm, which was already part of the FeatureIDE framework [KST+16]. The main contribution in this part is the generic explanation algorithm for defects in feature models. Our basic idea is founded on a successful application of BCP to the configuration process of an SPL [Bat05]. We extended the algorithm to deal with possible defects in a feature

model during its development and maintenance including dependencies between interrelated feature models. In combination with our user-friendly visualization, we argue that our approach is a novel achievement in the domain of automated feature model analysis. Overall, we reused, adapted and extended the idea of multiple feature models, a feature model slicing algorithm and an algorithm for the development of artificial intelligences to provide a successful answer to this research question.

We could already observe in our evaluation that the approach is not limited to just automation systems. In fact, we can apply the complete process to any kind of system given at least the feature model. This does also include industrial-size SPLs with thousands of features. The performance is only limited by the used SAT solver in the background. Although, SAT solving is proven to be NP-complete, the community has developed several heuristics and improvements to guarantee an efficient computation in many cases. Additionally, the core algorithm of BCP can easily be exchanged by other algorithms, which search for a minimal unsatisfiable set (MUS) to reduce the explanation length in the future. A swap from SAT to SMT solvers would also be imaginable.

> ***RQ2:*** *How can we capture the complete variability of a real-world automation system during its design time in a uniform way considering the solution space?*

First of all, the proposed multi-perspective modeling approach (cf. Chapter 4) is capable of capturing all relevant information of an automation system. The decision for UML models fosters an easy familiarization of developers as these models are well-known in mechanical, electrical and software engineering. While we focus on automation systems, a similar approach was developed for embedded systems in the SPES project [PHAB12]. We managed variability with the concept of delta modeling in a uniform way, since it is applied to all levels of abstraction in a likewise manner. These aspects were shown by successfully using variants modeled within our framework to control the PPU. The combination of UML models with delta modeling across three connected modeling perspectives concludes the answer for this research question.

Nevertheless, the presented approach leaves space for several points, which are worth of a continued discussion. We did include a prototypical consistency checking concept in our modeling framework. Yet, a literature review revealed that more sophisticated approaches to detect inconsistencies are available at least for other UML models, i.e., class and sequence diagrams [Egy06, RE12]. An application of these concepts to our UML models would be worth looking into. A similar observation can be made for automatic repair operations in UML models [LHE12]. In general, the application domain of our complete modeling framework is rather small by just considering automation systems. We argue that at least individual perspectives equipped with delta modeling can be reused in other domains as well,

since variability modeling in UML is itself a benefit. Finally, we only considered variability in space as all variants of the PPU were known in advance. However, automation systems evolve over time in order to fix defects or improve usability. Thus, the same implementation artifacts exists in different versions over time, the automation system evolves. This variability in time has to be addressed as well. Delta modeling is capable to capture both variability in space and time by the same means [Sch10, LKS16]. Hence, our multi-perspective modeling framework should be able to deal with evolution, but this aspect still needs further investigation within our proposed approach.

> ***RQ3:*** *How can we efficiently analyze product lines in the automation domain regarding performance metrics?*

Although this research question does not explicitly incorporate the necessity of being in the solution space, we stated in the introduction that performance analyses on feature models is insufficient for automation systems. Thus, the choice fell on design level models in the solution space, which we already developed in the previous contribution. Fortunately, the proposed workflow modeling perspective is able to manage variability with delta modeling. We further extended it with performance annotations to enable an actual performance analysis. By using the knowledge available in the deltas, we create an artificial family model representing the complete product line. Analyzing this model gave us the opportunity to efficiently derive performance metrics for individual variants without executing redundant computations. This family-product-based performance analysis is achieved by creating symbolic expressions to calculate the steady-state throughput at first and insert concrete performance values for individual variants afterwards. Overall, the efficiency and ultimately supremacy of our approach was shown in several experiments for both instantiations of the concept.

A point for discussion is how the performance values used for the annotation are obtained. In case of the PPU, we could rely on measurements of the physical machine. However, this option is not always available, especially if the system is still in development. We argue that this knowledge must be delivered by domain experts or a single initial simulation of the system to gain a basic understanding of the possible behavior. Limitations such as the missing fork/join barrier or evolution support are further discussed in the future work. Nevertheless, our proposed performance analysis is applicable to behavioral models of any workflow-type software system such as service-oriented architectures, data centers or systems where the focus is on control flow. Thus, a limitation to automation systems does not exist. Similar approaches are scarce in the literature as the performance community does not focus on system with high variability [GS11]. Family-product-bases analyses rely on the knowledge that all variants are known in advance. These analyses cannot deal with evolution,

e.g. the addition of a new variant to the product line, in an efficient way. The family model must be constructed from scratch again forcing us to redo the symbolic computations. We also limited ourselves to investigate the steady-state behavior, which may be insufficient if the system never reaches a steady-state. Both points have to be examined in the future.

By arriving at this point, we are finally able to provide an answer to the central research question, which asks the following:

> *How can we efficiently develop variant-rich systems in the automation domain and analyze their performance properties during design time?*

In this thesis, the efficient development relies on UML models equipped with delta modeling in the solution space. On top lies the feature modeling in the problem space to ensure a holistic software product line development process for automation systems. The analyze of performance properties during design time is based on the construction of a 150% model and a symbolic solution of the resulting performance equations. We believe that the observations taken from the individual evaluations and a comparison to related work justify the statement of being a novel result in the research community.

6.2 Future Work

While this thesis contributes approaches for variability modeling considering an interdisciplinary product line in both problem and solution space with good results, we envision future work to improve our current achievements. First, and foremost, all presented techniques have to be applied to additional real-world case studies. The best case would be a large-scale automation system in order to directly apply all steps of our improved automation product line development process. However, the variability modeling in the problem space as well as the efficient performance analyses can also deal with other SPL case study types as previously stated. Further case studies would allow us to generalize our observations.

Future Work for Interdisciplinary Variability Modeling. An explanation algorithm based on BCP has some disadvantages that should be addressed in the future. BCP does not ensure a minimal explanation and in rare cases it is not able to find an explanation at all [FK93]. These are obvious points for an improvement and a MUS algorithm is most likely suitable to compensate these problems [LS08]. The implementation of a MUS algorithm in FeatureIDE is straightforward possible, since it also operates on propositional logic and we can use a SAT solver to find the MUS.

Currently, we use BCP to explain defects occurring during the modeling phase of a feature model. However, BCP can also be applied to other aspects regarding an SPL, i.e., supporting developers during the configuration process of a concrete

variant. Preliminary work in this direction was already conducted in *GUIDSL*, but limited to the explanation of why certain features cannot be selected [Bat05]. A similar application would be the explanation of dead code, e.g., code fragments surrounded by an #ifdef statement which are never included in any feature combination [TSSPL09]. This problem requires a combined analysis of the feature model and implementation artifacts of the SPL. In particular, the conjunction of a presence condition, i.e., the formula containing feature selections which include the code block, and the feature model formula is not satisfiable [ABKS13]. Thus, BCP and MUS as well should be able to find an explanation.

Furthermore, we can exploit the available information in an explanation to actually fix a defect if this action is desired by the user. This functionality would be very similar to standard IDE features as available in Eclipse and Microsoft Visual Studio. All presented concepts concerning the visualization must also be extended to implicit constraints. At the moment, we only highlight the constraint and generate a textual explanation. This restriction should be lifted in the future. Furthermore, the slicing algorithm provides us with a non-editable feature model. Edit operations on this partial feature model are necessary, since we assume individual developers working on these models. Ultimately, the changes should be reflected back to the complete feature model to keep the models consistent and ease maintenance effort.

Future Work for Multi-Perspective Modeling. A sophisticated assessment of the usability of our multi-perspective modeling approach remains an open point. Thus, we would have to conduct a controlled user experiment with respect to understandability, learnability and maintainability of the modeling framework. Simultaneously, we can evaluate the improvement in usability of the graphical modeling approach over the textual one.

Although we have a prototypical consistency checking concept within our modeling framework, there is still room for improvement. The development of a consistency checking technique that takes full advantage of the available information in the deltas would be an adequate extension to maximize the support for developers during evolution and maintenance. Incrementality is key here, since it minimizes the response time as only changed parts and parts possibly affected by the changes need to be considered. As a basis for the incremental consistency checking approach, we envision work on type checking for delta-oriented product lines in Java [BDS13]. Other approaches available in the literature are also worth investigating such as work done in [Egy06, RE12, LHE10, LHE12].

Providing repair operations is a similar point as for the previous contribution. Again, if we are aware of an inconsistency, the knowledge can be exploited to provide developers with automated repair operations as in programming IDEs [MSS05, Egy06]. Finally, one can imagine to make the modeling framework runtime-aware [AGJ+14]. This would allow us to change the behavior, i.e, the cur-

rent variant of an automation system on-the-fly, and in addition, we can incorporate an efficient performance analysis at runtime. An application scenario would be a breakdown of one hardware part and based on the resulting performance metrics we can directly decide which would be the best alternative variant. Afterwards, we can apply the respective deltas and activate the variant on the machine.

Future Work for Efficient Performance Analysis in SPLs. We identify three major points regarding the performance analysis of variability-aware models. The first is about absence of "layering" in the performance models. Indeed, the class of queuing network models analyzed in this thesis are "flat", in the sense that there must be a distinction between the role of server and job in the system. Yet, it is well understood in the software performance community that in order to capture more realistic scenarios it may be necessary to also study "layered" models, e.g. [FAOW+09], where an entity (e.g., a web server) can be at the same a server (to incoming requests from a client) and a client (to perform requests to a database server). Other characteristics that deviate from the features of ordinary queueing networks are, for instance, fork/join behaviors. Yet, these are useful not only in traditional software systems, but also in the automation domain where it is not uncommon to find independent branches of a production line that merge, e.g., for the final assembly of a product.

The second point is about managing an evolving product line. The construction of a 150% model requires knowledge of all variants in advance. However, an automation system or any other SPL most certainly is subject to changes in the variants over time. The addition of a new variant or new changes in old variants force us to construct a new 150% model and re-compute the symbolic expressions necessary for the performance analysis, which is of course not desired and inefficient [TAK+14]. We envision an incremental performance analysis that is able to reuse previous computation results. In general, this may require us to move away from a family-product-based analysis strategy.

Finally, the third point is concerned with variability-aware reasoning techniques for transient dynamics. The family-product-based analyses are applicable in order to compute more efficiently the steady-state measures of performance. However, these techniques are oblivious to the behavior of the system in the transient state, i.e., when it is sufficiently away from a stationary regime. Thus, a runtime adaptation approach based on the above techniques may not be effective when the time to reach a stable solution is non-negligible. This calls for further techniques in order to adapt more promptly and dynamically to varying conditions.

Last but not least, the concept can always be applied to other queueing network types such as Gordon-Newell networks or networks with an Erlang distribution as service times [BGdMT05].

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013. 5, 10, 14, 163

[ACLF11a] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Decomposing Feature Models: Language, Environment, and Applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 600–603, Washington, DC, USA, 2011. IEEE Computer Society. 31, 69

[ACLF11b] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 424–427, Washington, DC, USA, 2011. IEEE Computer Society. 34, 35, 69

[AGJ+14] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2014. 102, 163

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. 72

[AKTS16] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. Implicit Constraints in Partial Feature Models. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 18–27, New York, NY, USA, 2016. ACM. 23, 29, 30, 31, 32, 33, 34, 45, 51, 52, 53, 54, 55, 70

[Amb08] Scott Ambler. *The Object Primer - Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2008. 72

[Ana16] Sofia Ananieva. Explaining Defects and Identifying Dependencies in Interrelated Feature Models. Master's thesis, TU Braunschweig, 2016. 33, 38, 40, 42, 43, 45, 48, 50, 56, 59, 60, 61

[Ant15] Anton Antonenko. Entwicklung und Evaluierung eines grafischen Modellierungsansatzes für delta-orientierte Aktivitätsdiagramme. Bachelor's thesis, TU Braunschweig, 2015. 95, 97

[Apt99] Krzysztof R. Apt. Some Remarks on Boolean Constraint Propagation. In *New Trends in Constraints*, pages 91–107. Springer, 1999. 39

[ATZ07] Danilo Ardagna, Marco Trubian, and Li Zhang. SLA based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing*, 67(3):259 – 270, 2007. 150

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20, Berlin Heidelberg, 2005. Springer. 16, 27, 28, 37, 38, 41, 65, 159, 163

[BBO+12] Steven Braun, Christian Bartelt, Martin Obermeier, Andreas Rausch, and Birgit Vogel-Heuser. Requirements on evolution management of product lines in automation engineering. In *International Conference on Mathematical Modelling (Mathmod)*, Vienna, Austria, 2012. 1, 2, 71, 159

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012. 10

[BDA+14] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: Unifying Class and Feature Modeling. *Software & Systems Modeling*, pages 1–35, 2014. 64

[BDIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310, 2004. 2, 104

[BDS13] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013. 163

[BFGR13] David Benavides, Alexander Felfernig, José A. Galindo, and Florian Reinfrank. *Automated Analysis in Feature Modelling and Product Configuration*, pages 160–175. Proceedings of the International Conference on Software Reuse (ICSR). Springer, Berlin, Heidelberg, 2013. 23, 28, 64

[BGdMT05] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley, 2005. 105, 108, 118, 120, 164

[BHLM13] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. Continuous approximation of collective system behaviour: A tutorial. *Performance Evaluation*, 70(5):317–349, 2013. 108

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. 2

[BM05] Simonetta Balsamo and Moreno Marzolla. Performance evaluation of UML software architectures with multiclass Queueing Network models. In *Proceedings of the International Workshop on Software and Performance (WOSP)*, pages 37–42, 2005. 2, 114

[BMB+15] Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, and Samuel Kounev. Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-Based Architectures. *IEEE Transactions on Software Engineering (TSE)*, 41(2):157–175, 2015. 109, 114

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 7:1–7:8, 2013. 4, 159

[Bro95] Frederick P. Brooks. *The Mythical Man-Month. Essays on Software Engineering*. Addison-Wesley Longman, 1995. 71

[BSBF11] Luca Bassi, Cristian Secchi, Marcello Bonfé, and Cesare Fantuzzi. A SysML-Based Methodology for Manufacturing Machinery Modeling and Design. *IEEE/ASME Transactions on Mechatronics*, 16(6):1049–1062, 2011. 100

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 187–197, 2003. 99

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010. 2, 3, 5, 15, 16, 23, 27, 28, 33, 36, 64, 159

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009. 67

[BSTRC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 129–134, 2007. 28, 65

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: a template approach based on superimposed variants. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 422–437, Berlin, Heidelberg, 2005. Springer-Verlag. 5, 99

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming*, 2011. 64

[CCS+13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE Transactions on Software Engineering (TSE)*, 99:1, 2013. 156

[CDI11] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011. 103, 104, 105, 106, 107, 108, 109

[CE05] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications: Methods, Techniques and Applications*. Addison-Wesley, 2005. 1, 2, 3, 9, 11, 14, 15, 16, 18, 23, 159

[CGS07] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 334–339. Springer, 2007. 67

[CHS+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2010. 156

[CL11] Marci Colla and Tiziano Leidi. Tools integration through a central model and automatic generation of multi-platform control code. In *Proceedings of the Workshop on Industrial Automation Tool Integration for Engineering Project Automation*, volume 821 of *CEUR*, 2011. 100

[CM02] Vittorio Cortellessa and Raffaela Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, 44(1):101–129, 2002. 2, 109, 115, 116, 117, 136, 154

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. 1, 10, 11

[Cop04] Ben Coppin. *Artificial Intelligence Illuminated*. Jones & Bartlett Learning, 2004. 37

[CP10] Dave Clarke and José Proença. Towards a Theory of Views for Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 91–98. Lancaster University, 2010. 67

[Cum82] Aldo Cumani. On the canonical representation of homogeneous Markov processes modelling failure-time distributions. *Microelectronics Reliability*, 22(3):583–602, 1982. 114

[DK86] Johan De Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, 1986. 37

[DLHE11] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. Cross-layer Modeler: A Tool for Flexible Multilevel Modeling with Consistency Checking. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 452–455, New York, NY, USA, 2011. ACM. 101

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Joint European Conferences on Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (ETAPS/TACAS)*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. 67

[DMI04] Antinisca Di Marco and Paola Inverardi. Compositional generation of software architecture performance QN models. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 37–46, 2004. 114

[Doy79] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, 1979. 37

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 39

[DPS12] Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A Formal Foundation for Dynamic Delta-oriented Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 1–10, New York, NY, USA, 2012. ACM. 102

[DS11] Ferruccio Damiani and Ina Schaefer. Dynamic Delta-oriented Programming. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 34:1–34:8, New York, NY, USA, 2011. ACM. 102

[Egy06] Alexander Egyed. Instant Consistency Checking for the UML. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 381–390, New York, NY, USA, 2006. ACM. 101, 160, 163

[Egy07] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 292–301, Washington, DC, USA, 2007. IEEE Computer Society. 101

[EPAH08] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Knowledge Based Method to Validate Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2008. 66

[EPAH09] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Using First Order Logic to Validate Feature Model. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 169–172, 2009. 64

[ER11] Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Journal of Information and Software Technology*, 53(1):2–13, 2011. 14

[ETCS14] Leire Etxeberria, Catia Trubiani, Vittorio Cortellessa, and Goiuria Sagardui. Performance-based Selection of Software and Hardware Features Under Parameter Uncertainty. In *Proceedings of the International Conference on Quality of Software Architectures (QoSA)*, pages 23–32, 2014. 138, 157

[FAOW+09] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering (TSE)*, 35(2):148–161, 2009. 119, 157, 164

[FBGR13] Alexander Felfernig, David Benavides, José Galindo, and Florian Reinfrank. Towards Anomaly Explanation in Feature Models. In *Proceedings of the International Workshop on Configuration*, pages 117–124. Citeseer, 2013. 66

[FDG08] Roman Froschauer, Deepak Dhungana, and Paul Grünbacher. Managing the Life-cycle of Industrial Automation Systems with Product Line Variability Models. In *Proceedings of the International Conference on Software Engineering and Advanced Applications (SEAA)*, Parma, Italy, 2008. 100

[FGT11] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 341–350, 2011. 156

[FK93] Kenneth D. Forbus and Johan De Kleer. *Building Problem Solvers*. MIT Press, 1993. 37, 38, 46, 162

[FLVH15] Stefan Feldmann, Christoph Legat, and Birgit Vogel-Heuser. Engineering Support in the Machine and Plant Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis. In *Proceedings of the International Symposium on Information Control in Manufacturing (INCOM)*, 2015. 29, 30, 31, 32, 34, 159

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering (FOSE)*. IEEE Computer Society, 2007. 72, 77

[Fri09] Andreas David Friedrich. *Anwendbarkeit von Methoden und Werkzeugen des konventionellen Softwareengineering zur Modellierung und Programmierung von Steuerungssystemen*, volume 2. Kassel University Press GmbH, 2009. 77

[FS10] Alexander Felfernig and Monika Schubert. Fastdiag: A Diagnosis Algorithm for Inconsistent Constraint Sets. In *Proceedings of the Workshop on the Principles of Diagnosis (DX)*, pages 31–38, 2010. 34, 66

[FTG15] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time. *IEEE Transactions on Software Engineering (TSE)*, 2015. 156

[GÏ5] Timo Günther. Enabling Graphical Variability Modeling for a Software System Across Three Layers of Abstraction. Bachelor's thesis, TU Braunschweig, 2015. 89, 94, 95, 97

[GCA+13] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 301–311, 2013. 155

[GDK+14] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Modeling the impact of workload on cloud resource scaling. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 310–317. IEEE, 2014. 149, 154

[GF99] Manish K. Govil and Michael C. Fu. Queueing theory in manufacturing: A survey. *Journal of Manufacturing Systems*, 18(3):214 – 240, 1999. 108, 138, 154

[GH04] Hassan Gomaa and Mohamed Hussein. Dynamic Software Reconfiguration in Software Product Families. In *Proceedings of the International Workshop on Software Product-Family Engineering (PFE)*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer Berlin Heidelberg, 2004. 157

[GKP+14] Sebastian Götz, Thomas Kühn, Christian Piechnick, Georg Püschel, and Uwe Aßmann. A Models@run.time Approach for Multi-objective Self-optimizing Software. In *Adaptive and Intelligent Systems*, volume 8779 of *Lecture Notes in Computer Science*, pages 100–109. Springer International Publishing, 2014. 102

[GM10] Yaser Ghanam and Frank Maurer. Linking feature models to code artifacts using executable acceptance tests. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag. 69

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. 5

[Gom06] Hassan Gomaa. Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures. In *Reuse of Off-the-Shelf Components*, volume 4039 of *Lecture Notes in Computer Science*, pages 440–440. Springer Berlin Heidelberg, 2006. 99

[GS11] Carlo Ghezzi and Amir Molzam Sharifloo. Verifying Non-functional Properties of Software Product Lines: Towards an Efficient Approach Using Parametric Model Checking. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 170–174, 2011. 2, 6, 156, 161

[GST16] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal unsatisfiable core extraction for SMT. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2016. 67

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. 75

[Hem08] Adithya Hemakumar. Finding Contradictions in Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 183–190, 2008. 64

[HGR12] Gerald Holl, Paul Grünbacher, and Rick Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *Information and Software Technology*, 54(8):828–852, 2012. 4, 32, 159

[HHS+11] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *Software & Systems Modeling*, 12(3):641–663, 2011. 68

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996. 119

[HKR+11a] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented architectural variability using MontiCore. In *Proceedings of the European Conference on Software Architecture (ECSA)*, pages 6:1–6:10, 2011. 75, 93, 99

[HKR11b] J. Happe, H. Koziolek, and R. Reussner. Facilitating Performance Predictions Using Software Components. *IEEE Software*, 28(3):27–33, 2011. 157

[HKRS05] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. *Consistency Problems in UML-Based Software Development*, pages 1–12. Proceedings of the International Conference on the Unified Modeling Language. Springer Berlin Heidelberg, 2005. 90

[HMPO+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Goran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 139–148, 2008. 4, 5, 99

[HTH13] Arnaud Hubaux, Thein Than Tun, and Patrick Heymans. Separation of Concerns in Feature Diagram Languages: A Systematic Survey. *ACM Computing Surveys*, 45(4):51:1–51:23, 2013. 3, 159

[IIS+11] Noraini Ibrahim, Rosziati Ibrahim, Mohd Zainuri Saringat, Dzahar Mansor, and Tutut Herawan. *Definition of Consistency Rules between UML Use Case and Activity Diagram*, pages 498–508. Proceedings of the International Conference on Ubiquitous Computing and Multimedia Applications (UCMA). Springer Berlin Heidelberg, 2011. 90

[ITT16] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. Symbolic Performance Adaptation. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*, 2016. 108, 149, 154, 157, 158

[Jac63] James R. Jackson. Jobshop-like Queueing Systems. *Management Science*, 10(1):131–142, 1963. 109, 111, 112, 118, 132

[JB08] Mikoláš Janota and Goetz Botterweck. *Formal Approach to Integrating Feature and Architecture Models*, pages 31–45. Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). Springer Berlin Heidelberg, 2008. 68

[JLM+12] Sven Jörges, Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen. A constraint-based variability modeling framework. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):511–530, 2012. 99

[Jun04] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2004. 65

[KA11] Christian Kästner and Sven Apel. Feature-oriented software development. In *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 346–382. Springer, 2011. 1, 2, 14

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, 2008. ACM. 4

[KAT16] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 132–143, New York, NY, USA, 2016. ACM. 23, 25, 26, 27, 34, 37, 38, 39, 40, 42, 43, 45, 46, 47, 48, 50, 57, 59, 60, 61, 65, 67, 70

[KATS17] Matthias Kowal, Sofia Ananieva, Thomas Thüm, and Ina Schaefer. Supporting the Development of Interdisciplinary Product Lines in the Manufacturing Domain. In *Proceedings of the World Congress of the International Federation of Automatic Control (IFAC)*, 2017. 23, 29, 30, 31, 32, 34, 36, 45, 51

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and Spencer Peterson A. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, 1990. 1, 2, 3, 15, 27

[KLL+14] Matthias Kowal, Christoph Legat, David Lorefice, Christian Prehofer, Ina Schaefer, and Birgit Vogel-Heuser. Delta Modeling for Variant-rich and Evolving Manufacturing Systems. In *Proceedings of the International Workshop on Modern Software Engineering Methods for Industrial Automation (MoSEMInA)*, pages 32–41, New York, NY, USA, 2014. ACM. 71, 72, 76, 78, 79, 80, 81, 83, 86, 88, 89, 96, 97

[KMLH11] Mohamed Khalgui, Olfa Mosbahi, Zhiwu Li, and Hans-Michael Hanisch. Reconfiguration of Distributed Embedded-Control Systems. *IEEE/ASME Transactions on Mechatronics*, 14(4):684 – 694, 2011. 100

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Proceedings of the Workshop on Feature Interactions in Telecommunications Networks and Distributed*, pages 284–297. IOS Press, 1997. 99

[KPST14] Matthias Kowal, Christian Prehofer, Ina Schaefer, and Mirco Tribastone. Model-based Development and Performance Analysis for Evolving Manufacturing Systems. *at - Automatisierungstechnik*, Volume 62(Issue 11):Pages 794–802, October 2014. 71, 81, 83, 86, 88, 89

[Kre16] Malte Kreutzfeldt. Inkrementelle Konsistenzprüfung in UML über mehrere Sichten. Bachelor's thesis, TU Braunschweig, 2016. 90, 91, 93

[KS00] Mohamed Mancona Kandé and Alfred Strohmeier. Towards a UML profile for software architecture descriptions. In *Proceedings of the International Conference on the Unified Modeling Language*, pages 513–527. Springer, 2000. 75

[KS16] Matthias Kowal and Ina Schaefer. Incremental Consistency Checking in Delta-oriented UML-Models for Automation Systems. In *Proceedings of the International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, volume 206 of *Electronic Proceedings in Theoretical Computer Science*, pages 32–45. Open Publishing Association, 2016. 71, 90, 91

[KSRB13] Dean Kramer, Christian Severin Sauer, and Thomas Roth-Berghofer. Towards Explanation Generation using Feature Models in Software Product Lines. In *Proceedings of the International Workshop on Knowledge Engineering and Software Engineering (KESE)*, 2013. 66

[KSS13] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards efficient SPL testing by variant reduction. In *Proceedings of the International Workshop on Variability & Composition (VariComp)*, pages 1–6, New York, NY, USA, 2013. ACM. 14

[KST14] Matthias Kowal, Ina Schaefer, and Mirco Tribastone. Family-Based Performance Analysis of Variant-Rich Software Systems. In *Proceedings of the International Conference on Fundamental Approaches to*

Software Engineering (FASE), volume 8411 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014. 103, 111, 112, 123, 124, 125, 129, 131, 147, 148

[KST+16] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing Algorithms for Efficient Feature-model Slicing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 60–64, New York, NY, USA, 2016. ACM. 32, 34, 35, 64, 69, 159

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614, Washington, 2009. IEEE. 28, 49, 57

[KTTS15] Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. Scaling Size and Parameter Spaces in Variability-aware Software Performance Models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, New York, NY, USA, 2015. ACM. 103, 119, 120, 122, 135, 139, 141, 150

[KTTS17] Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. Symbolic Performance Analysis of Variability-intensive Software Systems. 2017. submitted. 103, 116, 119, 120, 121, 122, 135, 139, 141, 142, 151, 153

[Kur70] Thomas G. Kurtz. Solutions of ordinary differential equations as limits of pure Markov processes. *Journal of Applied Probability*, 7(1):49–58, 1970. 119

[Lac17] Remo Lachmann. *Black-Box Test Case Selection and Prioritization for Software Variants and Versions*. PhD thesis, Technische Universität Braunschweig, 2017. 78

[LBKVH12] Fang Li, Gülden Bayrak, Konstantin Kernschmidt, and Birgit Vogel-Heuser. Specification of the Requirements to Support Information Technology-Cycles in the Machine and Plant Manufacturing Industry. In *Proceedings of the International Symposium on Information Control Problems in Manufacturing*, 2012. 71

[LEGP15] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. Feature modeling of two large-scale industrial software systems: Expe-

riences and lessons learned. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2015. 1, 3, 30, 32, 34, 65, 68

[LFVH13] Christoph Legat, Jens Folmer, and Birgit Vogel-Heuser. Evolution in industrial plant automation: A case study. In *Proceedings of the International Conference on Industrial Electronics Society (IECON)*, pages 4386–4391, 2013. 3, 19, 29, 30, 31

[LHE10] Roberto E. Lopez-Herrejon and Alexander Egyed. Detecting Inconsistencies in Multi-view Models with Variability. In *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA)*, pages 217–232, Berlin, Heidelberg, 2010. Springer-Verlag. 101, 163

[LHE12] Roberto E. Lopez-Herrejon and Alexander Egyed. Towards Fixing Inconsistencies in Models with Variability. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 93–100, New York, NY, USA, 2012. ACM. 101, 160, 163

[LKS16] Sascha Lity, Matthias Kowal, and Ina Schaefer. Higher-order Delta Modeling for Software Product Line Evolution. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 39–48, New York, NY, USA, 2016. ACM. 98, 161

[LLL+13] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based integration testing of large-scale systems. *Journal of Systems and Software*, 2013. 75, 78, 99

[LLVH13] Christoph Legat, Steffen Lamparter, and Birgit Vogel-Heuser. Knowledge-based Technologies for Future Factory Engineering and Control. In *Service Orientation in Holonic and Multi-agent Manufacturing and Robotics*, volume 472 of *Studies in Computational Intelligence*, chapter 23, pages 355 – 374. Springer, 2013. 100

[Lor14] David Lorefice. Delta-orientierte Modellierung und Konsistenzüberprüfung über mehrere Sichten. Master's thesis, TU Braunschweig, 2014. 78, 89, 93, 97

[LPKS14] Malte Lochau, Sven Peldszus, Matthias Kowal, and Ina Schaefer. Model-Based Testing. In *International School on Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014. 14

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40:1–33, 2008. 66, 162

[LSVH13] Christoph Legat, Daniel Schütz, and Birgit Vogel-Heuser. Automatic Generation of Field Control Strategies for Supporting (Re-)Engineering of Manufacturing Systems. *Journal of Intelligent Manufacturing*, 2013. 100

[LSW15] Uwe Lesta, Ina Schaefer, and Tim Winkelmann. Detecting and Explaining Conflicts in Attributed Feature Models. In *Proceedings of the International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 31–43, 2015. 28, 65

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 176–187, Berlin Heidelberg, 2002. Springer. 16, 65

[Mar09] Filip Marić. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009. 27

[MBC05] Johan Muskens, Reinder J. Bril, and Michel Chaudron. Generalizing Consistency Checking between Software Views. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 169–180, 2005. 90

[MBC+13] Marjan Mernik, Barrett R. Bryant, Giacomo Cabri, Maria Ganzha, Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming*, 78(6), 2013. 64, 69

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE Software*, 2003. 72

[MCMdO08] Marcílio Mendonça, Donald D. Cowan, William Malyk, and Toacy Cavalcante de Oliveira. Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis. *Journal of Software*, 2008. 68

[MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer-Verlag, Berlin Heidelberg, 2008. 23, 154

[Mey14] Claas Meyer. Graphische Darstellung von Deltas in UML-Statechart Diagrammen. Bachelor's thesis, TU Braunschweig, 2014. 94, 95

[MG15] Object Management Group. OMG Unified Modeling Language – Version 2.5. http://www.omg.org/spec/UML/2.5, June 2015. 5, 71, 73, 90, 93, 104

[Mit97] Isi Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1997. 113

[MS83] Joao P. Martins and Stuart C. Shapiro. *Reasoning in Multiple Belief Spaces*. PhD thesis, State University of New York at Buffalo, 1983. 37

[MSS05] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. *A Framework for Managing Consistency of Evolving UML Models*, pages 1–30. Software Evolution with UML and XML. IGI Global, 2005. 101, 163

[MTS+14] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An Overview on Analysis Tools for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 94–101, New York, NY, USA, 2014. ACM. 64

[NK08] Natsuko Noda and Tomoji Kishi. Aspect-Oriented Modeling for Variability Management. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 213–222, 2008. 5, 99

[NRS13] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 197–200, 2013. 67

[NSG+13] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 69–82, 2013. 149, 154

[Obj11] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)*. OMG, 2011. http://www.omg.org/spec/MARTE/1.1/. 109

[Oqu06] Flavio Oquendo. Formally modelling software architectures with the UML 2.0 profile for π-ADL. *ACM SIGSOFT Software Engineering Notes*, 31(1):1–13, 2006. 75

[Pan10] R. K. Pandey. Architectural description languages (ADLs) vs UML: a review. *ACM SIGSOFT Software Engineering Notes*, 35(3):1–5, 2010. 75

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin Heidelberg, 2005. 1, 3, 4, 9, 10, 11, 12, 13, 15, 23, 99, 159

[Pen03] Tom Pender. *UML Bible.* John Wiley & Sons, New York, USA, 2003. 73

[Pet13] Marian Petre. UML in Practice. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press. 73, 77

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology.* Springer Publishing Company, Incorporated, 2012. 5, 76, 99, 160

[PKHS10] Adam Piórkowski, Aleksander Kempny, Adrian Hajduk, and Jacek Strzelczyk. *Load Balancing for Heterogeneous Web Servers*, pages 189–198. Proceedings of the International Conference on Computer Networks. Springer Berlin Heidelberg, 2010. 150

[Pre04] Christian Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and Systems Modeling*, 3:221–234, 2004. 99

[RE12] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 220–229, 2012. 101, 160, 163

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Journal of Artificial Intelligence*, 32(1):57–95, 1987. 65

[RGMS14] Luisa F. Rincón, Gloria L. Giraldo, Raúl Mazo, and Camille Salinesi. An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 302:111–132, 2014. 29, 66

[RSS+07] Martijn N. Rooker, Christoph Sünder, Thomas Strasser, Alois Zoitl, Oliver Hummer, and Gerhard Ebenhofer. Zero Downtime Reconfiguration of Distributed Automation Systems: The εCEDAC Approach. In *Holonic and Multi-Agent Systems for Manufacturing*, Lecture Notes in Computer Science, pages 326–337. Springer, 2007. 100

[RSTS11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 11–22, NY, 2011. ACM. 3, 159

[Rut94] Vladislav Rutenburg. Propositional Truth Maintenance Systems: Classification and Complexity Analysis. *Annals of Mathematics and Artificial Intelligence*, 10(3):207–231, 1994. 37

[SBDT10] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag. 93, 99

[SCD12] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284, Berlin, Heidelberg, 2012. Springer-Verlag. 155

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 2006. 72

[Sch10] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 85–92, 2010. 4, 5, 17, 18, 81, 82, 98, 99, 102, 161

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 2003. 72

[SGAK15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence Models for Highly Configurable Systems. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 284–294, New York, NY, USA, 2015. ACM. 155

[SKK+12] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–177, 2012. 155

[SKT+16] Reimar Schröter, Sebastian Krieter, Thomas Tüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 667–678, New York, NY, USA, 2016. ACM. 34, 35, 65, 69, 70

[SLW12] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 252–268, Berlin Heidelberg, 2012. Springer. 68

[SRK+12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal*, 20(3):487–517, 2012. 64, 157

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. 72

[Ste09] William J. Stewart. *Probability, Markov Chains, Queues, and Simulation*. Princeton University Press, 2009. 105, 106, 107, 108, 109, 111, 113, 114, 115, 132

[STSS13] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. Automated Analysis of Dependent Feature Models. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 9:1–9:5, NY, 2013. ACM. 31, 67, 68

[SVZ11] Christoph Suender, Valeriy Vyatkin, and Alois Zoitl. Formal Validation of Downtimeless System Evolution in Embedded Automation

Controllers. *ACM Transactions on Embedded Control Systems*, 2011. 100

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001. 90

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):6:1–6:45, 2014. 2, 5, 6, 18, 19, 27, 67, 90, 91, 100, 125, 153, 155, 159, 164

[TBD+08] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software*, 81(6):883–896, 2008. 28, 64, 65, 66

[tBFGM15] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Using FMC for family-based analysis of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 432–439, 2015. 156

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264, Washington, 2009. IEEE. 3, 69

[tBLLV15] Maurice H. ter Beek, Axel Legay, Alberto L. Lafuente, and Andrea Vandin. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 11–15, 2015. 157

[TDGH12] Mirco Tribastone, Jie Ding, Stephen Gilmore, and Jane Hillston. Fluid Rewards for a Stochastic Process Algebra. *IEEE Transactions on Software Engineering (TSE)*, 38:861–874, 2012. 119

[TG08] Mirco Tribastone and Stephen Gilmore. Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In *Proceedings of the International Workshop on Software and Performance (WOSP)*, 2008. 109, 110

[TGH12] Mirco Tribastone, Stephen Gilmore, and Jane Hillston. Scalable Differential Analysis of Process Algebra Models. *IEEE Transactions on Software Engineering (TSE)*, 38(1):205–219, 2012. 119

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200, Washington, 2011. IEEE. 35, 69

[TLD+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. In *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*, page 2. ACM, 2011. 67

[TLG14] Damiano Torre, Yvan Labiche, and Marcela Genero. UML Consistency Rules: A Systematic Mapping Study. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 6:1–6:10, New York, NY, USA, 2014. ACM. 101

[TLG+16] Damiano Torre, Yvan Labiche, Marcela Genero, Maged Elaasar, Tuhin Kanti Das, Bernhard Hoisl, and Matthias Kowal. 1st International Workshop on UML Consistency Rules (WUCOR 2015): Post Workshop Report. *SIGSOFT Software Engineering Notes*, 41(2):34–37, 2016. 73, 75, 90

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the Conference on Computer Systems (EuroSys)*. ACM, 2011. 1, 16, 67, 155

[TP11] Rasha Tawhid and Dorina C. Petriu. Automatic Derivation of a Product Performance Model from a Software Product Line Model. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 80–89, 2011. 157

[TPK07] Kleanthis Thramboulidis, D. Perdikis, and S. Kantas. Model Driven Development of Distributed Control Applications. *International Journal of Advanced Manufacturing Technology*, 33(3-4):233 – 242, 2007. 72, 100

[Tri12] Pablo Trinidad. *Automating the Analysis of Stateful Feature Models*. PhD thesis, University of Seville, 2012. 65

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86. ACM, 2009. 67, 163

[UNThEs08] Muhammad Usman, Aamer Nadeem, Kim Tai-hoon, and Cho Eun-suk. A Survey of Consistency Checking Techniques for UML Models. In *Proceedings of the International Conference on Advanced Software Engineering and Its Applications (ASEA)*, pages 57–62, 2008. 100, 101

[UPS+05] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 291–302. ACM, 2005. 149, 154

[vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in Feature Models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004. 24, 25, 26, 64

[VG07] Markus Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 233–242, 2007. 4

[VHBKF11] Birgit Vogel-Heuser, Steven Braun, Benjamin Kormann, and David Friedrich. Implementation and Evaluation of UML as Modeling Notation in Object Oriented Software Engineering for Machine and Plant Automation. *Proceedings of the World Congress of the International Federation of Automatic Control (IFAC)*, 44(1):9151–9157, 2011. 77

[VHFF+15] Birgit Vogel-Heuser, Stefan Feldmann, Jens Folmer, Matthias Kowal, Ina Schaefer, Jan Ladiges, Alexander Fay, Christopher Haubeck, Winfried Lamersdorf, Sascha Lity, Timo Kehrer, Matthias Tichy, and Sinem Getir. Selected Challenges of Software Evolution for Automated Production Systems. In *Industrial Informatics (INDIN)*, 2015. 1, 71, 159

[VHLFF14] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit. Technical report, Institute of Automation and Information Systems, Technische Universität München, 2014. 19, 20, 21

[VHMK+15] Birgit Vogel-Heuser, Jakob Mund, Matthias Kowal, Christoph Legat, Jens Folmer, Sabine Teufl, and Ina Schaefer. Towards Interdisciplinary Variability Modeling for Automated Production Systems. In *Industrial Informatics (INDIN)*, 2015. 75

[VM10] Pavel Vrba and Vladimir Marik. Capabilities of Dynamic Reconfiguration of Multiagent-based Industrial Control Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 40(2):213 – 223, 2010. 100

[Vya13] Valeriy Vyatkin. Software Engineering in Industrial Automation: State of the Art Review. *IEEE Transactions on Industrial Informatics*, 9(3):1234 – 1249, 2013. 72, 100, 159

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. 63

[WVH04] Daniel Witsch and Birgit Vogel-Heuser. *Automatische Codegenerierung aus der UML für die IEC 61131-3*, pages 9–18. Eingebettete Systeme: Fachtagung der GI-Fachgruppe REAL-TIME, Echtzeitsysteme und PEARL. Springer Berlin Heidelberg, 2004. 77

[WVH11] Daniel Witsch and Birgit Vogel-Heuser. PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking. In *Proceedings of the World Congress of the International Federation of Automatic Control (IFAC)*, 2011. 100

[YVP13] Chia-Han Yang, Valeriy Vyatkin, and Cheng Pang. Model-driven Development of Control Software for Distributed Automation: a Survey and an Approach. *IEEE Transactions on Systems, Man and Cybernetics*, 2013. 100

[ZKDT11] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. Feature Assembly Framework: Towards Scalable and Reusable Feature Models. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 1–9, New York, NY, USA, 2011. ACM. 68

List of Abbreviations

ADL	=	Architecture Description Language
AIS	=	Institute of Automation and Information Systems
BCP	=	Boolean Constraint Propagation
BDD	=	Binary Decision Diagram
BE	=	Break-even Point
CAMUS	=	Compute All Minimal Unsatisfiable Subsets
CNF	=	Conjunctive Normal Form
CTC	=	Cross-Tree Constraint
CTMC	=	Continuous-time Markov Chain
DM	=	Delta Modeling
DSL	=	Domain Specific Language
DTMC	=	Discrete-time Markov Chain
EMF	=	Eclipse Modeling Framework
FB	=	Family-product-based Analysis
GEF	=	Graphical Editing Framework
GMF	=	Graphical Modeling Framework
IRS	=	Information Retrieval System
LAN	=	Local Area Network
LTMS	=	Logic Truth Maintenance System
MDD	=	Model-driven Development
MUS	=	Minimal Unsatisfiable Subset
ODE	=	Ordinary Differential Equation
PAAD	=	Performance-annotated Activity Diagram
PB	=	Product-based Analysis
PC	=	Pre-computation Time
PPU	=	Pick and Place Unit
QN	=	Queueing Network
SAT	=	Boolean Satisfiability Problem
SMT	=	Satisfiability Modulo Theories
SPL	=	Software Product Line
TM	=	Truth Maintenance
TMS	=	Truth Maintenance System
UML	=	Unified Modeling Language